Innovation Report

AUTOCODING METHODS FOR NETWORKED EMBEDDED SYSTEMS

Submitted in partial fulfilment of the Engineering Doctorate

By James Finney, 0117868

November 2009

Academic Supervisors: Dr. Peter Jones, Ross McMurran
Industrial Supervisor: Dr. Paul Faithfull

Declaration

I have read and understood the rules on cheating, plagiarism and appropriate referencing as outlined in my handbook and I declare that the work contained in this submission is my own, unless otherwise acknowledged.

Signed: …….James Finney

Acknowledgements

I would like to thank Rapicore Ltd and the EPSRC for funding this project. I would also like to offer special thanks to my supervisors: Dr. R.P. Jones, Dr. P. Faithfull, and R. McMurran, for their time, support, and guidance throughout this project.

Table of Contents

Figures

Tables

Definitions & Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASG	Abstract Syntax Graph
ASP	Active Server Pages
AST	Abstract Syntax Tree
CAN	Controller Area Network
CASE	Computer-Aided Software Engineering
CI	Continuous Integration
COGENT	COde GENeration Template
FIFO	First-In-First-Out
FILO	First-In-Last-Out
FlexRay	High-speed automotive network communication protocol
GCC	GNU Compiler Collection
Glade	User Interface designer for the GTK+ toolkit
GTK+	Open-source toolkit for creating graphical user interfaces
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HTML	Hyper-text Mark-up Language
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
Intelli-sense	Autocompletion implementation
JRE	Java Runtime Environment
LIN	Local Interconnect Network
MBD	Model-Based Design
MISRA	Motor Industry Software Reliability Association
NATO	North Atlantic Treaty Organisation
OEM	Original Equipment Manufacturer
PHP	PHP: Hyper-text Pre-processor
PHP-sat	Static code analysis tool for PHP
PHPUnit	Unit testing tool for PHP
SDF	System Description File
TDMA	Time Division Multiple Access
TLC	Target Language Compiler
UAT	User Acceptance Test
UML	Unified Modelling Language
W3C	World Wide Web Consortium
XML	Extensible Mark-up Language
XSLT	Extensible Stylesheet Language Transformations

Abstract

The volume and complexity of software is increasing; presenting developers with an ever increasing challenge to deliver a system within the agreed timescale and budget [1]. With the use of Computer-Aided Software Engineering (CASE) tools for requirements management, component design, and software validation the risks to the project can be reduced. This project focuses on Autocoding CASE tools, the methods used by such tools to generate the code, and the features these tools provide the user.

The Extensible Stylesheet Language Transformation (XSLT) based autocoding method used by Rapicore in their NetGen embedded network design tool was known to have a number of issues and limitations. The aim of the research was to identify these issues and develop an innovative solution that would support current and future autocoding requirements. Using the literature review and a number of practical projects, the issues with the XSLT-based method were identified. These issues were used to define the requirements with which a more appropriate autocoding method was researched and developed. A more powerful language was researched and selected, and with this language a prototype autocoding platform was designed, developed, validated, and evaluated.

The work concludes that the innovative use and integration of programmer-level Extensible Markup Language (XML) code descriptions and PHP scripting has provided Rapicore with a powerful and flexible autocoding platform to support current and future autocoding application requirements of any size and complexity.

1. Introduction

1.1. Background

The 1960s saw the start of the *Software Crisis*; a term coined by F. L. Bauer at the first NATO Software Engineering Conference in Garmisch, Germany. The term referred to the difficulty in writing correct, reliable and high quality software as computing power rapidly increased [2].

Since the invention of the integrated circuit in 1958, the number of transistors has increased exponentially; doubling approximately every two years. This trend was first observed by Gordon E. Moore, founder of Intel in his 1965 paper [3]. As the power of these devices increased, so too did the complexity of the problems and applications that could be tackled. The complexity of the software used to control these devices also grew through the increase in code volume, data, and control algorithm complexity required in more advanced applications. The complexity of the software not only challenged the developers in terms of designing, implementing, and validating the software; but also had serious implications on the management of the project and development process.

Through the increase in software complexity and the relative immaturity of software engineering as a discipline at the time, the crisis manifested itself in a number of ways: projects were unmanageable and therefore running over-budget and over-time; software was inefficient, of low quality, difficult to maintain, and often did not meet the requirements; software was never delivered [4].

The crisis encouraged software engineering research and resulted in a range of new methodologies, processes, tools, and other related disciplines. Software Engineering became more recognised as a discipline and is currently defined by the IEEE as *the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software* [5].

One of the most significant outcomes from the research was the range of CASE tools that can be used in projects to improve the reliability, quality, and speed of a software development. These tools covered all stages of a typical software development process such as: requirements capture; component design and simulation; software implementation; system validation [6].

Although CASE tools have been used successfully in many industries, ranging from automotive and aerospace to industrial and consumer; the continual increase in computing power and application complexity is still resulting in the same issues realised during the software crisis 50 years ago. A report by the Standish Group in 2004 found that only 29% of software projects in large enterprises produced acceptable results that were close to the agreed time and budget. Of the remaining 71%, 53% were significantly over budget and schedule, and 18% did not deliver any usable result. In addition, the projects outside the 29% had an average budget overrun of 56% [7].

Researchers in the field of Software Engineering, therefore, continue to develop new tools and improve existing tools to reduce the continuing impact of Moore's Law and software complexity on future software projects.

1.1.1. Autocoding CASE tools

As mentioned in the previous subsection, there are a range of CASE tools for the various stages in the software development process. The research in this project focused on one particular CASE tool which is used during the implementation stage of the software development process. These CASE tools are known as *Autocoders*, and are used to automatically generate an application's source code from the user's design – a process known as *Autocoding*.

Autocoding CASE tools provide a number of advantages for developers:

- *Speed of Development* – the source code can be generated from the design almost instantly;
- *Customisation* – the generated source code can be customised to meet the project's or developer's requirements;
- *Consistency* – changes in the software's design can be quickly and easily reflected in the software's implementation;
- *Design Focus* – developers can spend more time focussing on effectively solving the problem through the design, rather than on the solution's implementation.

In spite of the advantages, the widespread adoption of autocoding tools is limited for the following reasons:

- *People over Software* – the inability to communicate and discuss the various aspects of the software's implementation with a tool;
- *Tool Cost* – autocoding tools can have a high price tag, especially when these tools are aimed at critical applications such as aviation control systems;

- *Loss of Control* – the autocoding process is often black-boxed so the developer has no knowledge of the generation methods or the criteria used to transform the design into the implementation;
- *Code Bloat* – autocoders have been known to generate inefficient code. Although the tools are improving, many would still hesitate to invest based on this bad reputation.

Developers currently have a range of autocoding tools to choose from, the selection of which depends on the project's requirements and the developer's preferences. This range covers many types of applications which include: control algorithms, graphical data mapping, enterprise applications, code libraries, and device drivers to name a few. The fundamental operation of these autocoders are similar; all of them using a collection of inputs and specific techniques and methods to generate the required source code.

Every autocoder has one or more inputs and generates one or more outputs. These inputs and outputs are summarised in Figure 1.

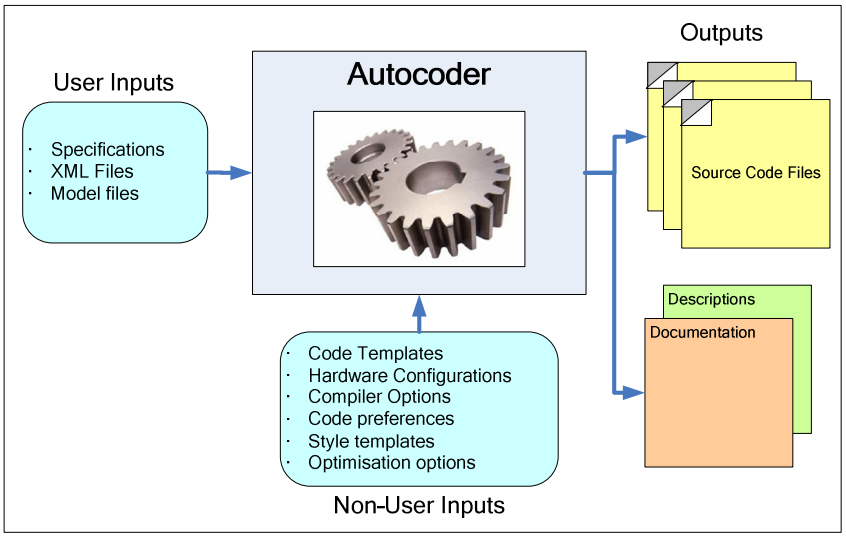

Figure 1 – An autocoder and its various inputs and outputs

1.1.1.1. Inputs

Autocoders automatically generate source code using a number of inputs which can be grouped into two types: *User Inputs* and *Non-User Inputs*. Autocoders use these two input types, along with the associated technologies and methods to generate customised source code for the user's application.

User inputs are those created by the user and are used by the autocoder to generate the source code. These inputs are usually created using a separate software tool and typically take the form of a text file containing the required input data. The input file describes the design of the software, or the collection of parameters and settings for the design. Common input file formats include XML files and proprietary file formats specific to a particular tool.

Non-user inputs are inputs that are used solely by the autocoder for the autocoding process. Most non-user inputs are dependent on the autocoder and the methods used, however, the majority of autocoders use some form of *code templates*. As the name suggests, code templates are plain text files that contain a *template* of the code to be generated. Each template is dedicated to one particular part of the output, and typically contains two content types: *Static Content*, which does not change throughout the autocoding process; and *Dynamic Content*, which is customised based on the user inputs.

1.1.1.2. Outputs

All autocoders generate one or more source code files. The source code is generated in a particular programming language and is customised using the data contained within the

user input files described previously. Some autocoders also allow the user to customise aspects of the code using a variety of options and settings (separate from the user input files). This additional customisation may take the form of: code formatting; identifier naming; code optimisation; programming language; target (processor).

How the source code generated is used by the developer, and what functionality the source code provides is again dependant on the autocoder used. Autocoders which use model-based design (MBD) inputs either generate: the application layer of the software application (such as control algorithms); the structural software components of a software application, such as those designed using the Unified Modelling Language (UML) [8]. The code templates for these autocoders represent the fundamental building blocks of the model, such as loops, conditionals, and classes. These tools could be referred to as *Generic* or *Application Layer Autocoders*.

Other autocoders focus primarily on the layers below that of the application layer (although it is possible for them to generate configurable application layers, i.e. which cannot be redesigned). These tools generate code libraries that are customised using the user's inputs and later integrated into the user's application code (created manually or using an Application Layer Autocoder). The code templates for these autocoders are less generic, and are larger in granularity; commonly representing an entire source code file. These tools could be referred to as *Specialised* or *Library Autocoders*.

In addition to generating source code, autocoders may also generate other files that are related to the inputs and the source code generated. The most common additional output is the source code's documentation, which can be kept consistent with the software's

design and implementation i.e. when the design or configuration changes; so too does the documentation.

1.1.1.3. Operation

The fundamental function of an autocoder is to use the various inputs to generate the source code. The exact methods and techniques used to achieve this are dependent on the autocoder in question and the inputs used. However, due to the unformatted textual nature of source code all autocoders primarily implement a range of string and text manipulation functions. Depending on the outputs required, the dynamic code sections may also perform calculations to determine the source code needed. Figure 2 shows a simple representation of how an autocoder uses the static and dynamic code templates to generate an output source code file.

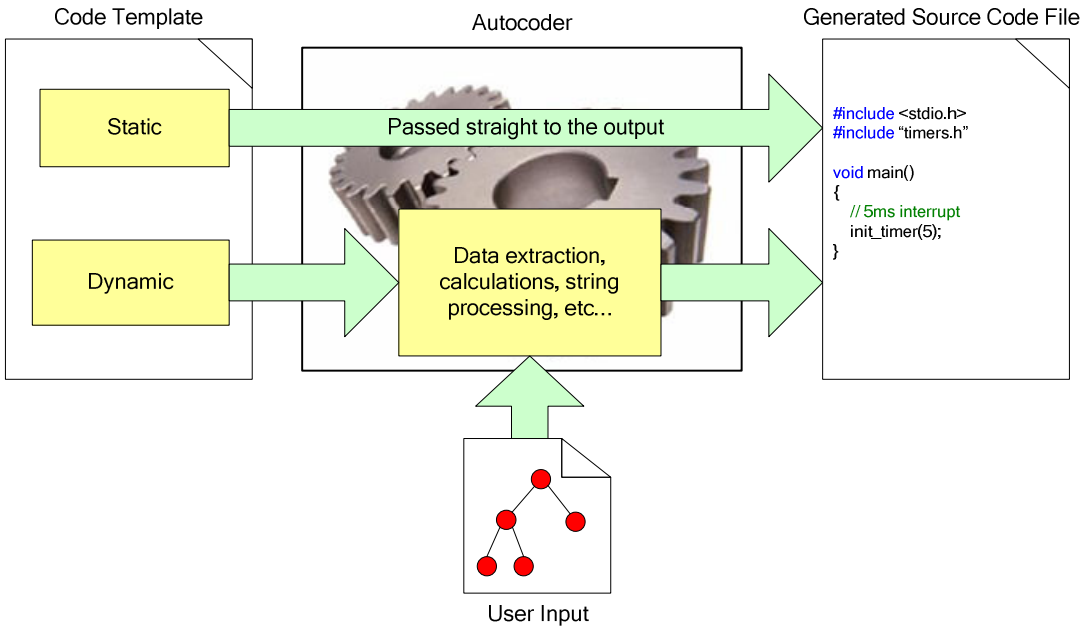

Figure 2 - Use of static and dynamic template types to generate source code

The static contents of the code templates; because they are not dependant on the user inputs, are passed directly to the output source code file (perhaps with some basic

formatting and manipulation). For the dynamic contents, the autocoder uses the user input file(s) to customise the output source code before it is output to the file. This usually involves calculations contained within the code templates themselves that calculate or create the code required.

1.1.2. NetGen

Rapicore, the sponsor of this project, is a specialist solutions provider for supporting the rapid prototyping of control systems and use techniques such as *automatic code generation*, hardware-in-the-loop simulation and safety engineering to provide solutions for their customers. Rapicore's main specialisation is in the area of embedded communication applications that use Controller Area Network (CAN), Local Interconnect Network (LIN), and FlexRay protocols [9]. Rapicore's main product, and the tool of focus for this research, is *NetGen*; an embedded network design and autocoding tool that allows distributed embedded system developers to design their networks and then automatically generate source code or related files for their network implementations.

NetGen is primarily aimed at the automotive industry, supporting the design and development of the three most common In-Vehicle Networking (IVN) protocols: CAN, LIN, and FlexRay. Commonly, one or more of these protocols are used within the vehicle; forming complex networks to support passenger entertainment, comfort, and safety. As the number of networked electronically controlled functions continues to increase, development tools such as NetGen can be used to ease the development process, reduce the product's time-to-market, and improve the overall robustness and reliability of the system.

Through NetGen's Graphical User Interface (GUI) shown in Figure 3, developers can configure their network and its constituent nodes, messages, schedules, and signals. Once the user has designed and configured their network, they can then use NetGen to automatically generate the source code for their application. NetGen could be considered a Library Autocoder and, unlike MBD and other autocoding tools, generates source code whose design and requirements are specified by the customer. The code templates are developed by Rapicore and combined with NetGen so that the customer can configure their network parameters and then generate their customised source code.

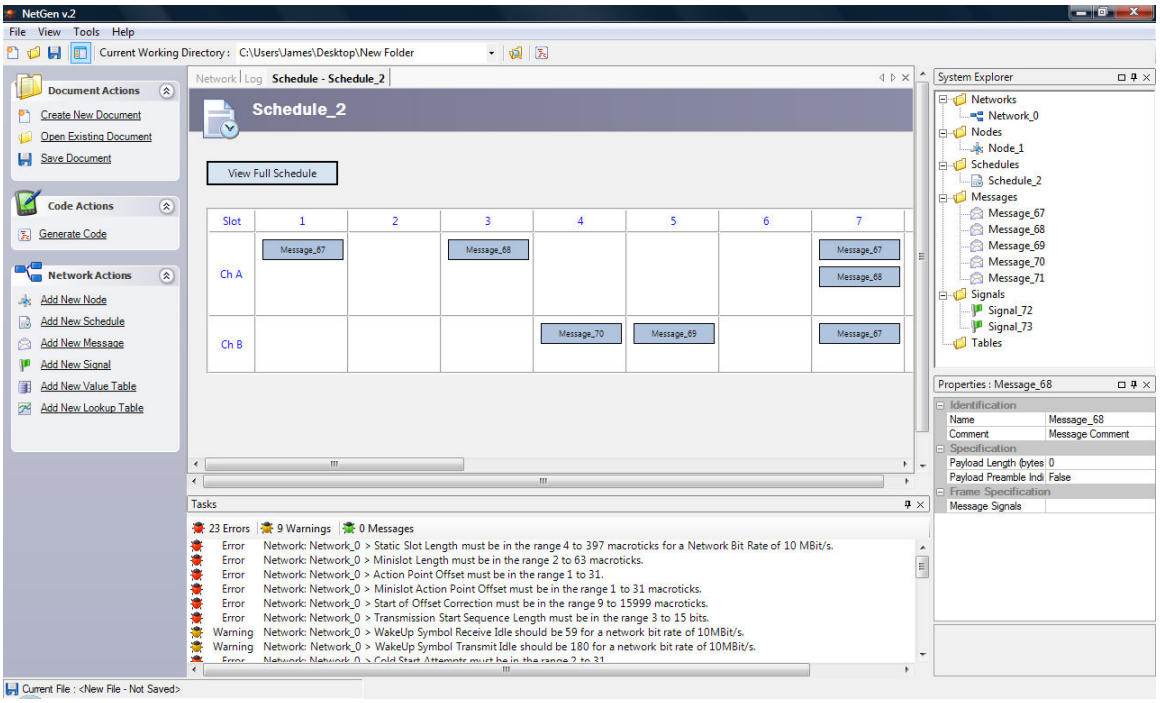

Figure 3 - NetGen Graphical User Interface

The source code typically generated by NetGen takes the form of communications stacks or device configuration header files. Rapicore also provides off-the-shelf signal-based Application Programmer Interface (API) stacks for each of the network protocols supported. These 'libraries' allow the user's application layer to communicate on an embedded network using only signals; abstracting away the underlying communication and network management processes.

NetGen's Autocoding Method

NetGen currently uses XSLT to generate the source code from the embedded network design. XSLT is an XML-based language which is used to transform XML documents into other XML and human-readable documents, and was originally developed by the World Wide Web Consortium (W3C) for use on the internet [10].

Referring to Figure 4, the XSLT processor takes two types of inputs: XML input files and XSLT stylesheets. The stylesheets contain a collection of template rules, instructions, and other directives to guide the XSLT processor in the production of the result document [11].

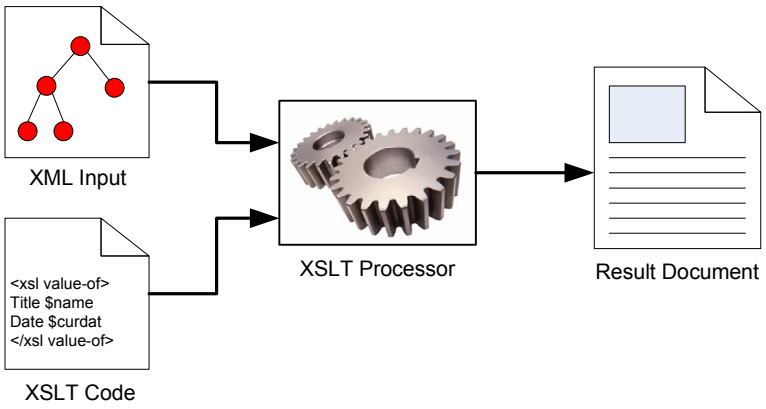

Figure 4 - XSLT document generation method

NetGen uses this same method to generate source code. The XML input is a System Description File (SDF); a proprietary file format which contains the network design data entered through the NetGen GUI. The XSLT stylesheets are the code templates, with there being one stylesheet for each source code file to be generated. This method provided a conceptually straightforward, ready to use autocoding method that was easily integrated into NetGen's implementation.

1.2. Research Aim and Methodology

1.2.1. Research Aim

The XSLT-based code generation method used by NetGen has been used successfully in a number of past projects. However, it was known that the method had a number of restrictive issues and limitations. These issues and limitations affected the output capabilities and the flexibility of the autocoding system, for example, the generated code was difficult to format consistently and the ability to tailor the code for a particular processor was not possible.

With autocoding being one of NetGen's core features it was vital that their autocoding methods and techniques were able to satisfy their customer's application requirements; most of which are provided directly by the customer. The autocoding method and associated infrastructure must be able to meet these requirements; not only to generate revenue through NetGen sales, but to also maintain Rapicore's reputation for delivering capable solutions.

The aim of the project was to prototype a new autocoding platform which utilised a more suitable autocoding method. To achieve this aim, the project had 3 objectives:

1. To research the requirements of autocoding tools in the context of Rapicore's core business, i.e. embedded network design and development, and the capabilities that such tools must provide its users;
2. To formally identify the issues and limitations of the current autocoding method in order to focus the research and development of an alternative;

3. To design, implement, and test a solution, and to validate that, both commercially and academically the autocoding tool and method have successfully met the stakeholders' requirements.

It is important to note that Rapicore's need for an alternative autocoding platform and method is to provide and improve upon the business's current autocoding capabilities. The new tool was not intended to be sold as a stand-alone product and to therefore directly improve upon the business's sales figures. This means that an increase in sales figures or revenue would not be used as a performance or success indicator for this project. However, in order to future proof the tool, and to not rule out the possible standalone retail of the tool once it has been made suitable for retail (see Future Work), the associated requirements have been considered at each stage of the prototype's research, design and development.

1.2.2. Research Methodology

The project adopted a logical and systematic research methodology to achieve the research aim. This methodology is reflected by the submissions in the portfolio and the information provided in this Innovation Report.

Figure 5 shows a diagram of the research methodology used. The 4 stages of the methodology and the activities performed in each stage are described below.

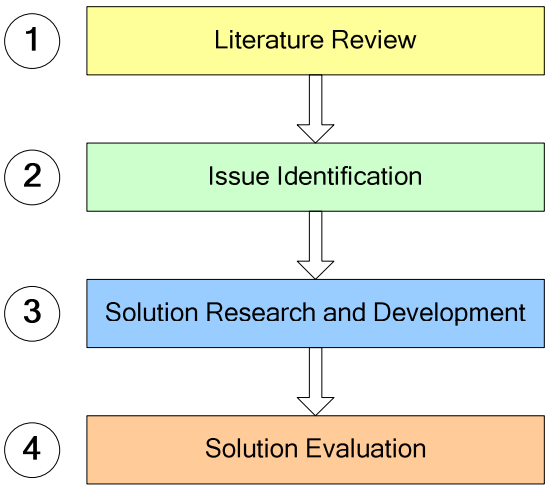

Figure 5 - Research Methodology diagram

1) Literature Review

The first stage of the research methodology was to undertake a comprehensive literature review. In addition to reviewing the literature, the literature review also covered the tools developed and used by other Original Equipment Manufacturers (OEMs), suppliers, and sectors, and the autocoding methods employed by these tools. The literature review aimed to provide knowledge and understanding of the following topics:

- Software engineering and software development;
- Software development issues that affect software projects;
- Current MBD and autocoding tools, and the features and benefits these provide;
- The technologies and methods used by current autocoding tools;
- Network design tools and the features and autocoding functionality these tools provide.

2) Issue Identification

Before a suitable solution could be developed, the issues and limitations with the XSLT-based method first needed to be identified. These were identified using the

following:

- The results of the literature review performed in Stage 1;
- Findings through practical experience with NetGen and its XSLT-based autocoding method.

Two development projects were undertaken that involved developing a network design, the XSLT code templates, and then generating the code required for the project. During these projects any issues, disadvantages, and limitations with the current implementation were recorded. The results of the literature review were also used to compare NetGen and the current autocoding method against existing tools and methods. The result was a collection of recommendations that would guide the research and development throughout the remaining methodology stages.

3) Solution Research and Development

For the third stage of the research methodology the recommendations documented in Stage 2 were transformed into a set of requirements. These requirements were then used to research and develop an innovative solution. The area of innovation was initially recognised as being associated with the methods used to generate the source code. This involved researching appropriate techniques and methods, and then developing a prototype autocoding platform to meet the research aim.

4) Solution Evaluation

Once an appropriate solution had been developed, the final stage of the research methodology was to evaluate the solution to ensure that the issues had been resolved and that the solution met Rapicore's engineering business requirements. This evaluation

used criteria based on the original issues identified in Stage 2 of the research methodology.

1.3. Portfolio and Innovation Report Structure

1.3.1. Portfolio Structure

The portfolio consists of 6 submissions numbered 1 through 6. This numbering corresponds to the intended reading order. These submissions and the research methodology stage under which each submission belongs are shown in Figure 6.

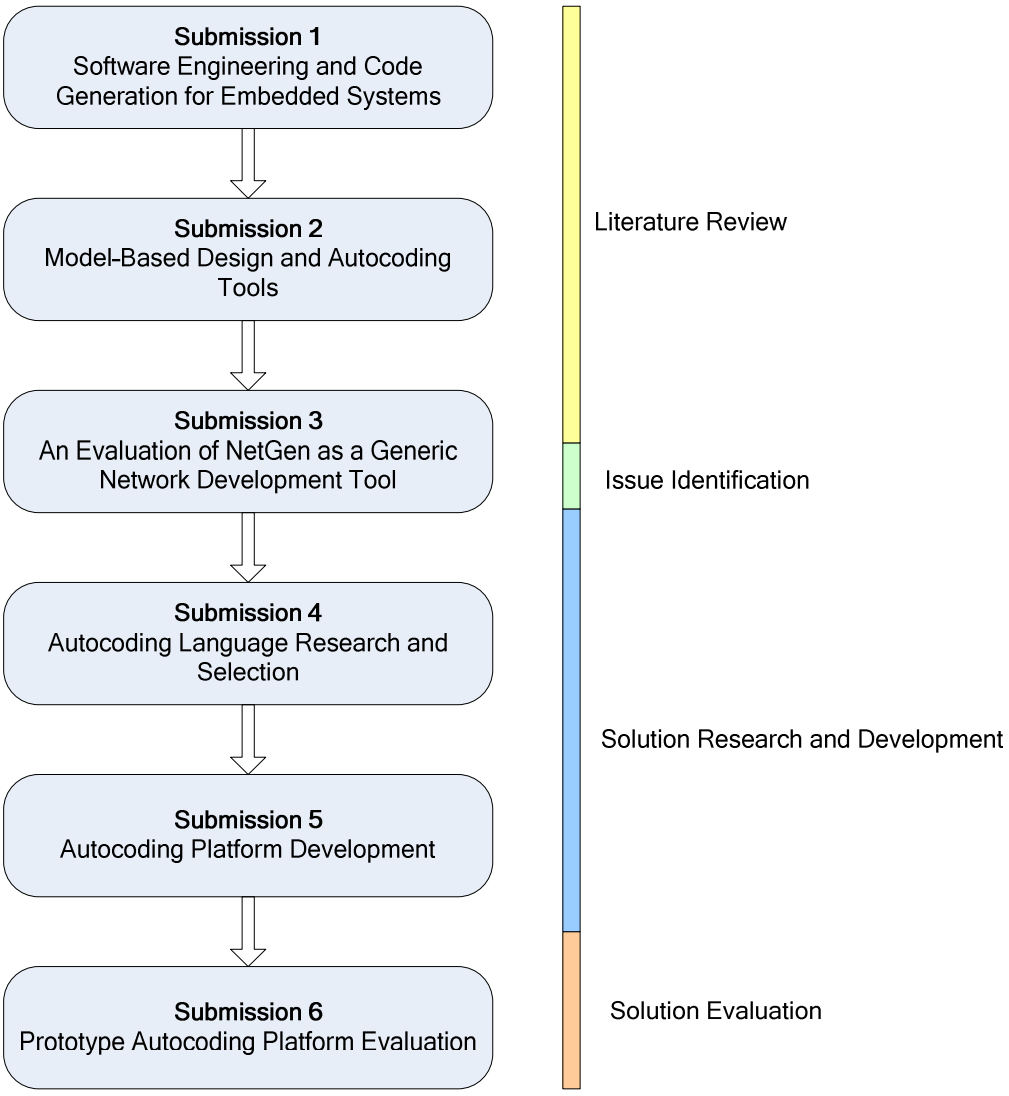

Figure 6 - Portfolio structure, submissions, and reading order

Submissions 1, 2, and the first half of submission 3 contain the project's literature review. Submission 1 reviews the wider scope of software engineering and software development, and discusses the issues experienced by developers and the basic operation of autocoding tools. Submission 2 reviews several popular MBD and autocoding tools used for embedded systems; covering their functions, features, and autocoding methods (where such information was available). The first half of Submission 3 overviews some competitive embedded network design tools and the features and autocoding functionality these tools provided.

The identification of the issues and limitations with NetGen's XSLT-based autocoding method is contained in the second half of Submission 3. The submission discusses 2 development projects undertaken; covering the aims, implementation, and results of each project. The submission concludes with a collection of recommendations that relate to the autocoding method used and the options and features provided to the user.
The research and development of a prototype autocoding platform, which aimed to solve the issues identified through a new autocoding method, is presented in Submissions 4 and 5. Submission 4 discusses the research and selection of a more suitable language for implementing the prototype autocoding platform. The submission covers the research methodology used to identify the most appropriate language in the context of this project, the details of the research process, and the results. Submission 5 describes the development of the prototype autocoding platform using the selected language. The document describes the selection of an appropriate development methodology and the software's requirements definition, design, implementation, and validation.

The final submission (Submission 6) presents the evaluation of the prototype autocoding platform. The document begins by defining and describing the criteria used for the evaluation. Following this, the document presents a case study that tests out the autocoding methods implemented by the platform and discusses the results of the evaluation.

1.3.2. Innovation Report Structure

The structure of this innovation report follows the research methodology and the associated submissions described above.

Section 2 continues this innovation report by presenting the main findings from the literature review. Section 3 discusses the issue identification process and the recommendations made. Section 4 then overviews the research and development of the prototype autocoding platform. Section 5 overviews the evaluation of the prototype autocoding platform and the results. Section 6 contains the main discussion, where the claim of innovation is stated and the innovation is described and justified. The penultimate section (Section 7) concludes the research project and the final section (Section 8) discusses the areas of future work.

2. Literature Review

NetGen used an XSLT-based method for automatically generating the source code from the user's network design, as described in Section 1.1.2. Although this method had been used successfully in a number of past projects, it was known to Rapicore that this method had a number of issues and limitations. These issues and limitations had not been identified, quantified, or documented at the time, and the functions and features expected by a user from such an autocoding application had not been fully researched and subsequently implemented.

The literature review focused on the autocoding tools and methods currently used by others. Based on the knowledge available at the time, the literature review had 3 primary aims:

1. To understand the current state of autocoding research and development;
2. To understand the current functions and features provided by autocoding tools;
3. To understand the current methods and techniques used for the autocoding process.

The literature review was divided into two stages. The first stage of the literature review was to develop an understanding of software engineering, software development, and other associated topics. This provided the basic knowledge required for the main literature review that followed. The literature review for the first stage covered the following topics, all of which are presented in Submission 1 of the portfolio [12]: software engineering; software development models; CASE tools; code generation; manual coding and human factors; autocoder applications and operation.

Once a basic knowledge was acquired, the second stage of the literature review focussed on the options provided by current autocoding and network design tools, and the current methods used for the autocoding process.

This section will discuss the second stage of the literature review and its findings. Section 2.1 will discuss the autocoding options provided by the MBD and network design tools reviewed. Section 2.2 presents the findings from the autocoding method research. Finally, Section 2.3 will summarise all of the autocoding option and method findings that were used later in the project.

2.1. Autocoding Options

From previous personal experience with NetGen and the XSLT-based autocoding method, it was known that the tool provided few options that allowed the user to customise the source code generated from their network design. This, in most cases, would result in generated source code that did not entirely suit their application and its requirements, for example, the code could not be customised to suit a particular processor or networking hardware.

One part of the literature review, therefore, was to investigate the options provided by other autocoding tools available. The findings from the literature review could then be used in the design and implementation of the solution to ensure that these options were provided.

The investigation of autocoding options covered two categories of tools: MBD tools and Network design tools. The reasons for reviewing these tools, summaries of the tools reviewed, and the resulting findings are contained within this subsection.

2.1.1. Model-Based Design and Autocoding Tool Review

MBD is a mathematical and visual method of developing complex control systems. MBD tools allow developers to graphically construct their models using various libraries of visual blocks, which can be connected together as required. The developer can provide stimulus to the model and simulate their designs; observing the system as it is simulated to ensure that the design functions correctly [13].

MBD tools were reviewed during the literature review due to their common integration with autocoding tools. In this context, the autocoders are used to automatically generate the source code from the user's model. The MBD tool review covered 3 tools: *Simulink* [14] (The MathWorks), *SCADE* [15] (Esterel Technologies), and *ASCET-MD* [16] (ETAS). These tools were chosen in particular due to their: use for developing embedded systems, like NetGen; their popularity and widespread use throughout industry.

Autocoder	Company	Model-based Design tool	Description
Real-Time Workshop [18]	The MathWorks	Simulink	Generates ANSI C source code
Stateflow Coder [19]	The MathWorks	Stateflow	Generates ANSI C source code from a Stateflow diagram
TargetLink [20]	dSPACE	Simulink	Generates ANSI C
SCADE Code Generators [21][22][23]	Esterel Technologies	SCADE Suite	Consists of 3 separate autocoders for 3 standards: DO-178B, IEC 61508, EN50128
ASCET-SE [24]	ETAS	ASCET-MD	Generates C source code from ASCET-MD model

Table 1 - Model-based design oriented autocoding tools reviewed

The literature review covered 5 autocoders, which were chosen due to their specific integration with the MBD tools also reviewed. The autocoding tools reviewed are summarised in Table 1. The full review of the MBD and associated autocoding tools can be found in Submission 2 [17].

The literature review for the MBD and associated autocoding tools used product datasheets and the tools' websites, as well as a number of papers where developers had used these tools in their applications. In addition, due to the availability of Simulink and TargetLink, an example autocoding model was created to gain practical experience of a 3rd party tool.

2.1.2. Network Design and Autocoding Tool Review

Embedded network design tools can be used by developers to define and specify the network parameters of their application. Some tools also allow the developer to simulate and test their network designs without having the hardware present.

Embedded network design and autocoding tools were reviewed to match the intended market of NetGen. The network design review covered 9 tools in total; some of which were network design tools only; some were autocoders only; and some provided both network design and autocoding functionality. The tools reviewed are summarised in Table 2. The particular network design tools reviewed were selected due to their focus on embedded networking protocols such as CAN, LIN, and FlexRay. These matched the network protocols supported by NetGen.

As with the model-based design and autocoding tool review, the literature for the review came from the tools' datasheets and websites. The full network design and autocoding tool review can be found in the first half of Submission 3 [29].

Tool	Developer	Description
Design and Analysis		
Volcano Network Architect	Mentor Graphics [25]	CAN and LIN
DaVinci	Vector [26]	System, network, and data design
CANoe	Vector	Development, testing, and analysis of ECUs and networks with simulation and emulation of ECUs
Tresos Designer	Elektrobit [27]	System design and configuration
TTX-Plan	TTTech [28]	Network design and automatic scheduling tool for distributed automotive systems based on FlexRay.
LIN-Plan	TTAutomotive	Design and development of LIN networks.
Code Generation		
Volcano Target Package	Mentor Graphics	Precompiled object libraries and associated documentation
DaVinci	Vector	
Tresos Studio & AutoCore	Elektrobit	AUTOSAR module configuration and module generation
TTX-Build	TTAutomotive	Modular configuration tool for AUTOSAR components.

Table 2 - Embedded network design and autocoding tools reviewed

2.1.3. Findings

The literature review included 15 MBD, network design, and autocoding tools from 9 companies. The review covered the autocoding functions and features provided by these tools, such as: optimisations, code customisation, language support, processor and compiler support, documentation generation, and standard support. A summary of the review's main findings is provided below.

Optimisations

ASCET-MD, SCADE, Simulink and TargetLink allow the user to select their desired level of source code optimisation. These optimisations allow the developer to perform a

trade-off between various properties of the code and its implementation such as: code size; processing speed; and RAM (data) size. ASCET-MD, Simulink and TargetLink provide the greatest level of optimisation as they can tailor the code for specific processors, allowing the generated code to take advantage of the specific hardware resources available.

Code Customisation

SCADE, Simulink and TargetLink allow the user to customise the non-functional form of the source code generated. This customisation covers code formatting (indentation, vertical spacing, and whitespace) to improve the visible structure and readability of the code. Users can also customise the identifier naming styles used for items such as variables and functions, for example, the use of Camel Case (e.g. MyVariable) or Pascal Case (e.g. myVariable) for identifiers. Of the tools reviewed, TargetLink provided the greatest level of code customisation, enabling the user to generate source code that better met a developer's or organisation's coding standard.

Code Language Support

All of the MBD autocoders shown in Table 1 were able to generate source code in ANSI C; mainly due to the popularity of this language in embedded applications. SCADE also allowed users to generate source code in other languages including Qualifiable C, Ada, and Spark Ada. Ada in particular is a common language for safety-critical embedded applications due to its type safety.

Processor and Compiler support

VTP, Tresos, DaVinci, ASCET-SE, Simulink, and TargetLink provide code generation and optimisation for specific embedded processors. This functionality is often provided through separate modules that are sold separately from the main MBD or autocoding tool. In addition to this, it was found that the autocoders for the network design tools generated pre-compiled code (in contrast to NetGen which generates non-compiled source code). This is due to the philosophy of AUTOSAR (see *AUTOSAR Support* below) and the need for automotive developers to protect their intellectual property in such a development environment.

Code documentation generation

All of the tools reviewed provide automated documentation features that document the generated code and link this code to the input model. The documentation provides information such as the code generation options and the optimisations that were used to generate the code. This enables developers to repeat previously used model and code generation configurations for the same or other similar models.

Coding standard support

ASCET-MD and SCADE can generate source code that adheres to a variety of coding standards or guidelines. These standards and guidelines include: DO-178B, IEC 61508, EN50128, and MISRA C. MISRA C is particularly popular in the automotive industry and DO-178B is used in the aerospace industry. Tools which support the safety-related standards such as DO-178B (e.g. SCADE) are often expensive due to the autocoder's certification requirements.

AUTOSAR Support

It was clear from the network design tools covered that the *Automotive Open Systems Architecture* (AUTOSAR) standard has a strong influence on the tool itself and the intended purpose of the autocoder.

AUTOSAR is a recent automotive software standard developed by a number of industrial partners including BMW, Ford, Toyota, and Volkswagen. The main aims of AUTOSAR are to: standardise automotive software; provide the scalability of software to different vehicle platforms and variants; increase the use of off-the-shelf software components in vehicles. The standard specifies a number of different software layers and components with well defined interfaces so that they can be easily bought in by the development team and connected together without modification [30].

Network design and autocoding tools are of great benefit to developers where AUTOSAR is concerned as the standard is large and complex. There are a large number of individual software modules present in the standard with each having well defined interfaces, interactions, and configurations. Due to AUTOSAR's standardisation it is important that all of the modules match the AUTOSAR requirements, and with a design tool tailored as such this standard conformance can be better guaranteed.

2.2. Autocoding Methods

There are various methods and techniques for automatically generating source code; however, most of these methods are based on the same basic principles as described in the introduction. In order to develop a suitable and innovative solution an awareness of the methods currently available needed to be acquired. The aim of the autocoding

method literature review, therefore, was to investigate the autocoding methods currently being researched and used to aid the research and development of a solution later in this project.

The literature review so far has focused on tools aimed at embedded systems. These were chosen to match the intended end-user of Rapicore's NetGen tool. The autocoding method review investigated the wider field of autocoding, in addition to those methods used by the MBD and network design tools reviewed. This was necessary in order to increase the scope of literature and methods available, for example, with the exception of the method used by The MathWorks, most of the tools reviewed did not disclose this information.

During the literature review, two main areas where identified as being the focus of current autocoding method research. The first area covered autocoding methods that related to the generation of code for compilers. These methods focus on the more mathematical aspects of software and low-level optimisations which are beyond the scope of this research project. The second area covered autocoding methods that focused on the high-level, template-oriented autocoding methods. The second area was the focus for this part of the literature review.

This subsection will present the findings from the autocoding method literature review performed. Specifically, the subsection will cover: code template implementation (2.2.1); autocoder front-ends (2.2.2); processing (2.2.3). Note that some of the findings presented here are not contained within the portfolio. After the initial literature review

was performed, some additional research was seen as necessary for the purposes of this research project.

2.2.1. Code Template Implementations

The code templates, as mentioned in the introduction, contain a template of the code to be generated. These code templates are used together with the input files to generate the required source code.

Source code templates are typically written in a particular scripting language. These script-based templates extract the required data from the input data source, apply it as necessary to the template and then output the result.

One specific method found during the literature review allowed template developers to combine dynamic scripts with the static contents of any ASCII formatted text file. This method was implemented by a tool called *CodeSmith Studio*, which accepts templates written in a language similar to ASP.NET and may contain scripting code in C#, VB.NET, or JScript [30]. A brief example of a CodeSmith Studio template is shown below.

```
<%@ CodeTemplate Language="C#" TargetLanguage="HTML" %>
...
<html>
  <head>
    <title>Test Report</title>
  </head>
  <body>
    <h1><%= TestReport.ProjectName %></h1>
    <h2>Test Date: <%= TestReport.TestDate %></h2>
  </body>
</html>
```

The items contained within the '<%=' and '%>' tags contain the dynamic script written

in C#, VB.NET, or JScript. The dynamic scripts contained within these tags are first parsed by the CodeSmith processor. The processor then replaces the original tag with the result of the evaluated script.

CodeSmith Studio also provides a GUI front-end which aids the user in developing the templates required for their applications. This will be discussed further in the following subsection.

2.2.2. Autocoder Front-Ends

Most of the autocoding tools researched during the literature review provided the user with a front-end. The intended purposes of these front-ends include:

- The collection of input data from the user (rather than having an input file);
- Allowing the user to configure the autocoding process;
- Aiding the developer during the template development process.

An example of a front-end which is used for capturing the user's input data is a tool intended for transforming Software Design Patterns into source code implementations [32]. This tool consisted of a Web Browser front-end (Figure 7) for capturing the user's inputs and responding to user events (such as the clicking of a button).

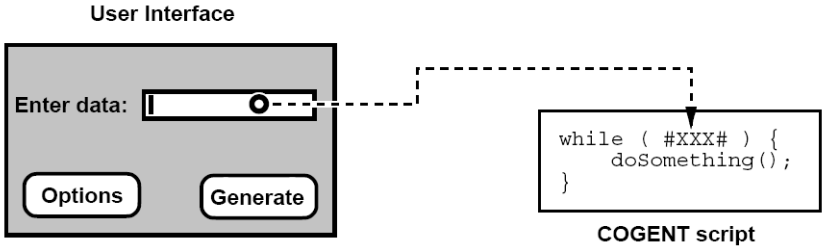

Figure 7 – Mapping an input value to a COGENT parameter [32]

Once the required information has been captured, the Web Browser then invokes a Perl-based mapper that maps the user's data and selections to the appropriate code templates for the required design pattern. The mapper then passes that information to a custom processor called *COGENT* (COde GENeration Template), which then generates the source code for the design pattern selected. This process and the flow of control is shown in Figure 8.

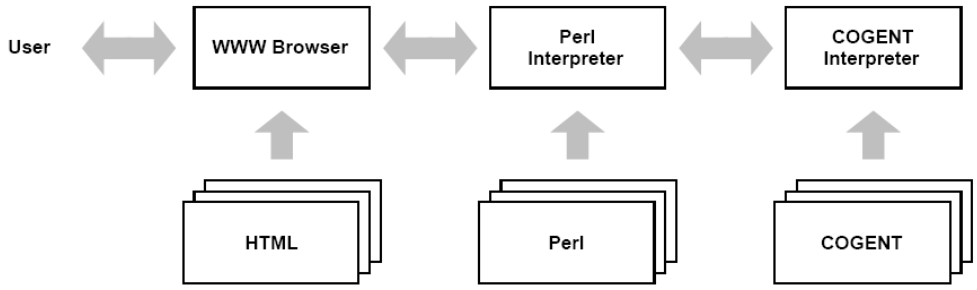

Figure 8 – Implementation of the design pattern tool [32]

In terms of front-ends for aiding the template development process, there was CodeSmith Studio who's template implementation was discussed in Section 2.2.1.

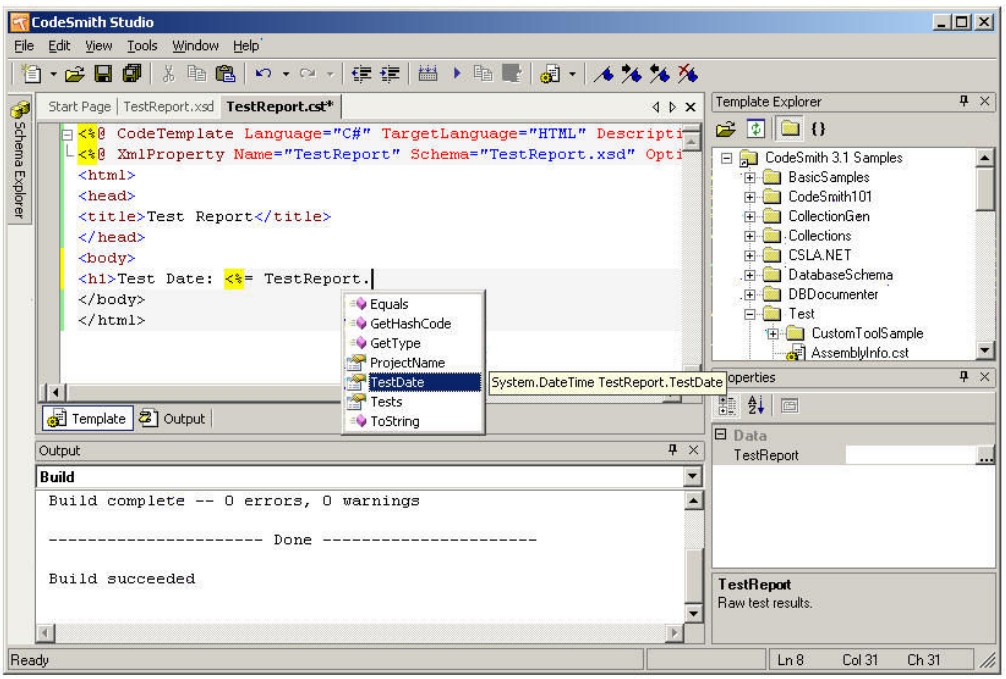

Figure 9 – The CodeSmith Studio graphical user interface

The CodeSmith Studio front-end (Figure 9) provides features that are similar to those provided by Microsoft's Visual Studio [33]; the most obvious being that of IntelliSense. IntelliSense monitors the text written by the user in real-time and provides a list of available functions and data that can be used in the current context.

In addition to IntelliSense, CodeSmith also provides syntax highlighting, project navigation, and syntax checking to further increase the productivity of the developer.

2.2.3. Processing

All autocoders reviewed are based around one or a number of existing scripting languages. The use of existing scripting languages allows companies to develop autocoding tools using existing interpreters, which have already been thoroughly tried and tested by other developers. In addition, the use of scripting languages themselves, as opposed to compiled alternatives, provides an easier to use and compatible processing method for ASCII text files such as source code templates.

Autocoders that implement existing scripting languages use the script processor for the language in question. For example, Java-based autocoders use the Java Runtime Environment (JRE) to process the Java code templates. Similarly, XSLT-based autocoders (such as NetGen's method) use the XSLT processor to process the XSLT-based code templates.

In terms of the processing performed in the code templates, it was found that there was a mixture in the mathematical and computational complexity involved. Some tools used simple text replacement in the code templates and little computational processing, such

as the Design Pattern tool discussed previously, and demonstrated in Figure 7. In this instance, the code templates contain a mark-up which is later replaced by the data. Other tools provide, or could potentially provide, a large amount of dynamic computations, an example of which being the code templates that can be developed using CodeSmith Studio.

It was noted that some tools customise or borrow principles (such as syntax) from an existing scripting language, e.g. CodeSmith Studio, whose syntax is very similar to that of ASP.NET. Unfortunately, many high-value tools such as the MBD and autocoding tools reviewed do not disclose the methods used unless such knowledge is required by the user to customise the autocoding output.

Real-Time Workshop and TargetLink are examples of tools that provide information on how the high-level processor works (however, the underlying operation is not disclosed). These products used a propriety tool called the *Target Language Compiler*, or TLC. The TLC is part of the Mathwork's Real-Time Workshop code generator and transforms a specially compiled Simulink block diagram into ANSI C source code.

The TLC is an integral part of Real-Time Workshop, enabling the user to customise the C code generated from any Simulink model. Through customisation, users can produce platform-specific code, or they can incorporate their own algorithmic changes for performance, code size, or compatibility with existing methods. An overview of the TLC process is shown in Figure 10.

After reading in the `model.rtw` file, the TLC generates its code based on; *target files*, which specify particular code for each block; *model-wide files*, which specify the overall code style. The TLC works like a text processor, using target files and the *model.rtw* file to generate ANSI C code. To create a target-specific application, Real-Time Workshop also requires a template *makefile* that specifies the appropriate C compiler and compiler options for the build process.

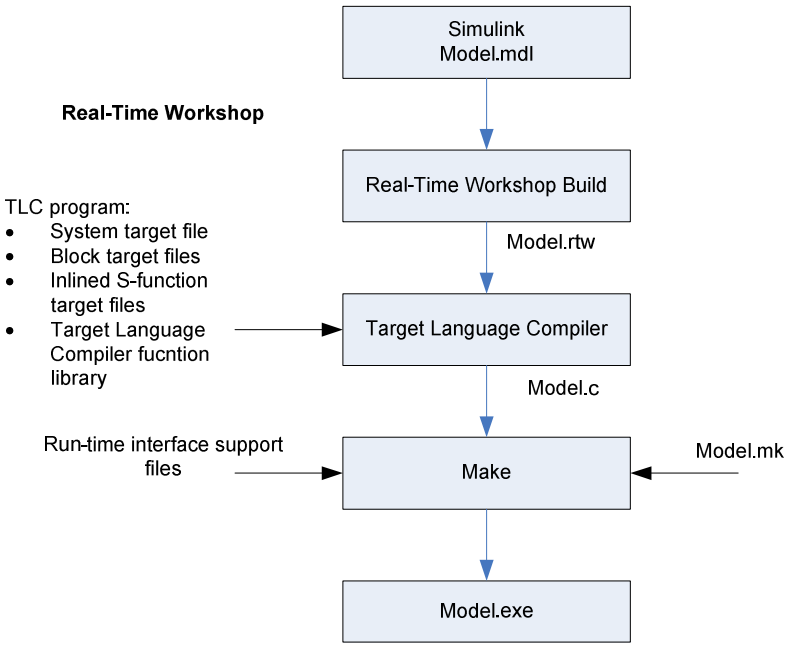

Figure 10 - Overview of the TLC process

The TLC uses a mark-up syntax similar to that of the HyperText Markup Language (HTML) and XML, along with the power and flexibility of Perl and other scripting languages, and the data handling functionality of MATLAB to go from the model file to source code. Through practical use it was found that TargetLink also used TLC, but with the addition of XSLT files to add code styling functionality to the tool.

2.3. Summary

The main part of the literature review focused on the autocoding tools and methods currently used by others. The literature review had 3 primary aims:

1. To understand the current state of autocoding research and development;
2. To understand the current functions and features provided by autocoding tools;
3. To understand the current methods and techniques used for the autocoding process.

The observations made during the literature review are summarised in the following subsections. These findings were later considered during the NetGen analysis (Section 3) and the prototype autocoding platform's development (Section 5) to ensure that a suitable and effective solution was created.

2.3.1. Model-Based Design and Autocoding Tools

Table 3 shows a summarised comparison of the MBD and autocoding tools reviewed (see Appendix A for the full comparison table).

	ASCET-MD	SCADE	Simulink	TargetLink
Code Optimisation	Yes	Yes	Yes	Yes
Embedded Support	Yes	No	Yes (with Embedded Coder)	Yes
Specific Processor Support	Yes	No	Yes (with Embedded Target)	Yes
Code Customisation	Unknown	Yes	Yes	Yes
Standards/Guidelines	MISRA-C IEC 61508 SIL3	D0-178B IEC 61508 EN 50128 MISRA-C	None	None
Code Languages	ANSI C	Ada & Ada Spark C and Qualifiable C	ANSI C (Standard), Any other language using S-Functions	ANSI C (Standard), Any other language using S-Functions

Table 3 - Comparison of the MBD and autocoding tools reviewed

Autocoding tools allow the user to select their desired level of code optimisation. This, for example, enables the developer to trade-off code size for processing speed and vice versa. Tools that also provide specific target support such as Simulink and TargetLink are able to provide further levels of optimisation; tailoring the code to make more effective use of the hardware resources available. It is worth noting that although SCADE is the only MBD tool that does not provide this processor specific, the reason for not providing such support is due to the tool's certification requirements for standards such as DO-178B.

SCADE, Simulink and TargetLink allow the developer to customise the formatting and style of the generated source code (ASCET-MD may have provided this, but this could not be confirmed). Code formatting includes: whitespace, horizontal spacing, and new lines. Style refers to the naming styles used for identifier names such as variables and functions. Such customisation enables developers to generate code that is easier to read and better meets a company's coding standards; helping minimise the need for manual code modification post-generation.

All four of the MBD and autocoding tools can generate source code in ANSI C. SCADE, Simulink and TargetLink also allow the user to generate source code in other languages such as Ada. This functionality widens the tools market; allowing developers to generate source code that matches their development and application requirements.

ASCET-MD and SCADE support a number of coding standards and guidelines such as MISRA C, DO-178B, and IEC 61508. The adherence to these standards and guidelines

can help ensure that the code generated is more robust and reliable for use in the intended, often safety-critical applications.

2.3.2. Network Design Tools

The network design tool review found that, as well as providing design entry and validation, the tools also allowed developers to export their designs as one or more network description files such as FIBEX (Field Bus Exchange) and LDF (LIN Description File). These files contain the configuration of the nodes, messages, signals, and schedules within the network, and allow developers to easily communicate and transport their designs between other tools and parties involved in the system's development.

The three most common network protocols supported by current design tools are: LIN, used primarily for automotive body control; CAN, the most widely implemented protocol of the three, which is used for a range of functions in many industries; and FlexRay, a high-speed fault tolerant protocol. Although these can be used in a number of applications from aerospace to automation, they are primarily used in automotive control systems.

With all of the tools reviewed being focused on in-vehicle network development, they all provided support for the AUTOSAR standard. The tools allow developers to configure and automatically generate AUTOSAR software that can be used in their applications.

From the networking tools reviewed, it was found that all of them generated pre-

compiled object libraries, rather than source code. This links in with the AUTOSAR functionality (and the protection of Intellectual Property (IP)), and therefore makes code customisation (in terms of formatting and styling) irrelevant for these tools.

2.3.3. Autocoding Methods

The autocoding methods research investigated code template implementations, autocoding front-ends, and code processing. The review included both experimental and commercial tools such as CodeSmith Studio, COGENT, and TLC.

The research found that tools such as CodeSmith Studio and COGENT in particular can effectively use GUIs; not only start the autocoding process, but to: collect additional information from the user; aid the development of the code templates.

All of the autocoding tools researched are based on scripting languages; commonly using existing scripting languages to a greater or lesser extent. Those tools that use existing scripting languages to a lesser extent often add additional functionality to existing languages or borrow concepts from a particular language.

One of the main elements of the autocoding methods used is the addition of custom mark-up to the code templates. These mark-ups are either used for simple text replacement, i.e. the replacement of the mark-up with the value required, or they were found to contain scripts so that computations could be performed within the template.

CodeSmith Studio, COGENT, and TLC rely on multiple languages and techniques to automatically generate the source code required. For example, TLC made use of Perl,

XML, and XSLT for various aspects and functions within the tool. The combination of multiple languages allows the autocoder to benefit from the key functions and features provided by a variety of languages and technologies, e.g. XML for data storage and representation, and Perl for processing.

CodeSmith Studio and TLC allow the users to develop their own code templates or customise existing code templates to suit their application. This ability means that such tools can be customised to meet the developer's requirements, allowing them to add further autocoding functionality that is not provided off-the-shelf.

3. NetGen Analysis

The literature review discussed in the previous section resulted in a list of options offered by current autocoding and network design tools. These covered items such as code optimisations, language and processor support, and code documentation generation. The literature review also investigated the methods and techniques currently used for the autocoding process, and covered template implementations, front-ends, and processing.

With the functions and features available in other autocoding tools now known, the next stage of the project involved the analysis of NetGen and the XSLT-based autocoding method to identify the issues and limitations. The aim of this analysis was to provide a list of recommendations based on the issues and missing options identified. These recommendations would then be used to define the requirements of a more suitable and capable autocoding method.

It was decided that the most effective way of identifying the issues and missing features was through practical experience with NetGen and the XSLT autocoding method. Therefore, two practical projects were undertaken which involved adding autocoding support for the CAN and FlexRay embedded network protocols. These projects were representative of a typical autocoding development; providing the same functionality and adhering to the same or similar requirements and design of previous projects undertaken before starting this project. This section will discuss the aims and results of these two projects (Sections 3.1 & 3.2), and then present the autocoding method and option recommendations defined (Section 3.3). Full details of the NetGen analysis, the

projects undertaken and corresponding discussion can be found in portfolio Submission 3 [29].

3.1. CAN Project

3.1.1. Aims

Prior to the CAN project, NetGen supported the configuration of CAN networks but did not provide CAN autocoding functionality. The aim of the CAN project was to add this support to NetGen. The code to be generated from NetGen had a number of requirements defined by the company and typical customer requirements. These requirements were:

- To provide a signal-based interface for the user's application;
- To abstract the low-level CAN implementation away from the user;
- To support three bit formats for the packing and unpacking of signals within CAN messages (Motorola Forward, Motorola Backward, Intel);
- To generate code specifically for the NXP (formally Philips) ARM7 LPC2129 32-bit Microcontroller [34].

3.1.2. Results

One of the main problems found during the CAN project was with the generation of code using XSLT. As mentioned in the introduction, XSLT was developed by the W3C, and was intended for transforming XML documents into other XML and human-readable documents. The main problems found when using such a technique for autocoding during this project are summarised below:

Lack of Mathematical Functions

XSLT only supports the basic mathematical functions such as addition, subtraction, and multiplication. The more complicated functions such as bitwise operations and trigonometry (to name a few) are not available for use in the code templates by the developer, limiting XSLT's use to simple autocoding applications.

Lack of Computational Functions

In addition to a lack of mathematical functionality, it was found that XSLT also lacked many other functions that may be required for developing code templates. These functions cover string manipulation, text processing, and file system access, to name a few. This again limits the complexity of the autocoding applications that such a system can support.

Text Formatting

It is important that any code generated by an autocoder is well structured, so that it can be easily read, understood, and used in the user's application. This structure refers to items such as whitespace, vertical spacing, and indentation, and their appropriate use to help define the code's structure.

Unfortunately, XSLT makes it difficult to intentionally align and structure the text within the output file. This is especially difficult in more complicated dynamically generated sections; with the process often reducing itself to one of trial and error.

Lack of Variable Support

Variables are used within code templates, as they are in source code, to keep track of data as the code is processed. One such example that was required during the CAN project in particular was a simple variable that incremented during each iteration of a loop. This variable is then often used as an index into an array, or alternatively for simply outputting to the generated file.

Although XSLT does have data type called a 'Variable', its use and functionality is highly restricted. It can have a local or global scope like a variable, but its value cannot be changed once it is first assigned a value. This means that the variable is of very little use; functioning more like a constant than a true variable.

3.2. FlexRay Project

3.2.1. Aims

The aim of this project was to add code generation support for the latest FlexRay networking protocol. This formed part of a larger collaborative project called SAPECS and consisted of three other companies: Atmel Semiconductors, Valeo Engine Management, and Ayrton Technologies (GeenSys as of 2007), all based in France. The aim of SAPECS was to develop a FlexRay demonstration that showed the use of FlexRay in future vehicle control systems. Rapicore's position in the project was to provide NetGen so that the collaborators could configure the FlexRay parameters and generate configuration code files for the network controllers. The requirements of the code was set by the SAPECS project collaborators, and consisted of example code files that needed to be generated by the tool and customised using their configurations entered through NetGen's GUI.

3.2.2. Results

Unfortunately, the code generation required for this project was a lot more complicated and required many features that were unsupported by XSLT. The majority of these issues were common to the CAN project discussed previously. This meant that the project could not be completed using the XSLT-based autocoding method: the user could configure the network, but no code could be generated for the project.

One of the largest contributing factors to the failure of the FlexRay project concerned the computational and mathematical processing power of XSLT, as highlighted in the CAN project. The code required for the FlexRay project was very complex in places and required a lot of processing before any output could be generated. It quickly became apparent that this task was not possible with the XSLT method used by NetGen. Some of the difficulties, in addition to the ones already mentioned in the CAN project included:

No cross template variables

Often two code files are dependent on one another, i.e. the code generated in one file depends on the code generated in another. XSLT provided no means of saving information and transferring it between different code templates.

No 2D Array Variables

The FlexRay implementation required two 2D arrays for calculating the message transmission and reception events required by each processor on the network. Each array represented one of the two channels in the FlexRay network, and each index-able location in the array represented a transmission or reception time slot (based on a Time

Division Multiple Access (TDMA) scheme). XSLT had no array data types (single or multi-dimensional) meaning that these calculations could not be performed, and therefore the messaging events could not be implemented in the required processors.

The lack of data type support was not limited to arrays either. XSLT also lacked many other data types which include other aggregates in addition to arrays, such as: classes, unions, structures, or enumerators. Again, the lack of these data types severely limited the functionality that could be implemented in the code templates.

3.3 Recommendations

3.3.1. Autocoding Method

It was clear that, due to the fundamental limitations of the language which were demonstrated during the CAN and FlexRay projects, the XSLT autocoding method used by NetGen needed to be changed in order to support any future application requirements. It was decided that the research and development of a more suitable and capable alternative method should consider the following recommendations. These recommendations were that the method needed to satisfy the following:

1. It needed to provide more power in terms of :
    a. Mathematical functions and computation;
    b. Inherent functionality and constructs;
    c. Output formatting capabilities.
2. It needed to be flexible in terms of:
    a. The possible amount of template granularity;
    b. Output configurability;
    c. Template reusability.
3. It needed to be compatible with NetGen in terms of:
    a. Its ability to work as a standalone deployment;

b. Its ease of integration with NetGen considering its current structure;

4. It needed to be cost effective in terms of:
    a. The amount of money required to use or purchase any technologies required;
    b. The amount of time required to develop and/or implement the technology to generate code.

3.3.2. Autocoding Options

From the main literature review presented in Section 2, it was found that one or more of the autocoding tools reviewed provided the following code generation options that were currently not provided by NetGen:

- The optimisation of generated code, such as the trade-off between execution speed and code size;
- Multiple-target support and the integration of this support with code optimisation;
- User-specified code styling and identifier naming conventions;
- ANSI-C compliant code generation – no checks are done at present in NetGen to ensure ANSI-C compliance (although compliance is not explicitly stated at any point);
- The optional generation of code in different languages such as Ada or C++ - although this is theoretically possible in NetGen, no options exist to select between languages;
- The optional adherence to particular standards and guidelines such as MISRA C or IEC 61508;
- The integration with third party tools such as compilers, MBD, and requirements capture tools;
- Automated documentation generation.

It was recommended that the solution should investigate ways of providing the following specific options to the user, based on the above observations:

- Support for particular targets (processors);
- Compiler selection;
- Code language selection;
- Code compliance selection;
- Code styling and formatting customisation;
- Compilation of generated code;
- Generation of code documentation.

4. Prototype Platform Research and Development

The previous section presented the list of autocoding method and option recommendations that resulted from both the literature review and NetGen analysis undertaken. Using these recommendations the research project was then in a position to research and develop a capable and suitable solution.

It was clear from the work done and the associated findings that XSLT and the existing autocoding system could not be used or improved upon to achieve the aims of the project. Therefore, the research and work focused on the development of a new 'Prototype Autocoding Platform' for NetGen based around a more suitable language. The platform was intended as a functional base with which the autocoding methods and techniques chosen could be fully implemented and tested.

The research and development activity was split into three stages:

1. The research and selection of a more appropriate language for implementing the autocoding process and platform;
2. The development of the prototype autocoding platform;
3. The evaluation of the prototype.

This section will discuss the first two stages of the prototype autocoding platform's development listed above. The first subsection (4.1) will describe the language research and selection process performed, covering the methodology used, the research performed, and the testing and selection process. The second subsection (4.2) describes the prototype autocoding platform's development, covering the development model used, and the software's requirements definition, design, implementation, and validation.

4.1. Language Research and Selection

The Interactive Roster of Programming Languages compiled by Murdoch University contains information on 8512 programming languages [35]. The goal of the research methodology was to gradually narrow down the large population of languages to the single, most appropriate language over a number of stages. A summary of the research methodology used to achieve this is shown in Figure 11.

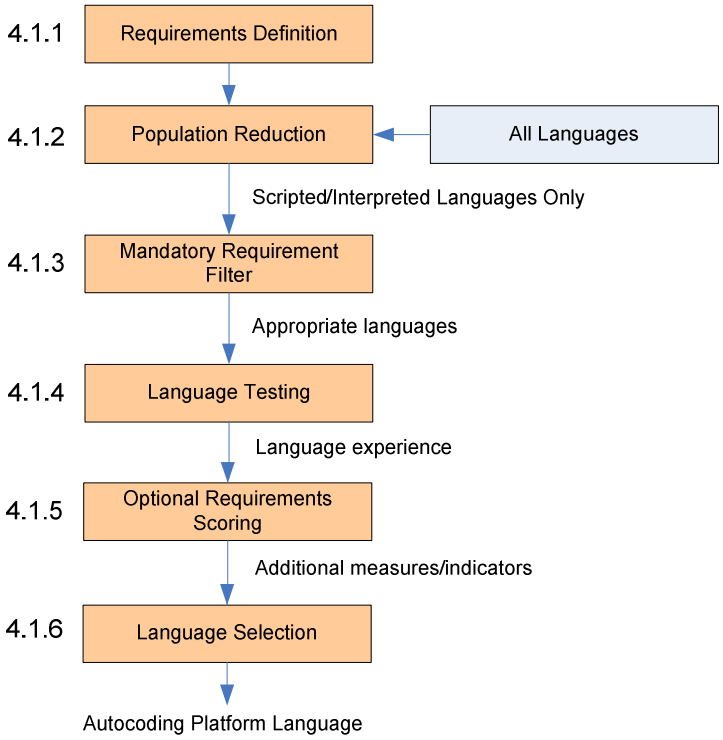

Figure 11 - Language selection Research Methodology

Each stage of the research methodology shown in Figure 11 is described in the subsections that follow.

4.1.1. Requirements Definition

A set of requirements for the new language were defined using the proposed autocoding method recommendations listed in Section 3.3. Two categories of requirements were

defined for the selection process: *Mandatory Requirements*, which were used to judge a language's 'fitness for purpose' and needed to be satisfied; *Optional Requirements*, which were used to compare languages that satisfied the mandatory requirements so that a better decision could be made. The requirements defined for the alternative language are summarised in Table 4.

Req. #	Requirement Description:
	Mandatory: The alternative technology must:
1	be available for the Windows operating systems (OS)
2	be a general-purpose programming language
3	be available as a standalone deployment, without requiring any other application/s installed to run
4	provide string/text oriented processing
5	provide XML parsing, as well as other file reading capabilities
6	provide the ability to create, edit, and delete files and directories, as well as being able to check for the existence of files and directories
7	provide control over output formatting including indentation, newlines, and number formatting
8	support 'true' variables and array data types
9	provide additional mathematical and other computational functionality over that of XSLT
10	support variables at a global scope between templates
11	support run-time configurability
12	support low template granularity
13	provide the ability to feedback progress and status information back to NetGen
14	provide adequate language and function documentation
15	be cost effective to use or implement
16	be easy to integrate with NetGen
17	support object-oriented programming
18	have development stability
	Optional: It would be beneficial for the alternative technology to have/provide:
19	a C-like programming style
20	good literature and community support
21	a small distributable size
22	good code readability
23	dynamic typing

Table 4 - Mandatory and optional requirements for language selection

The mandatory requirements were arranged into an approximate order of importance, and each language was analysed from the most important to the least important; stopping and moving onto the next language once a language failed to satisfy a requirements.

4.1.2. Population Reduction

Due to the large number of languages available, it would have been infeasible to apply the requirements to all of them; therefore the large population of languages first needed to be reduced. To achieve this, only interpreted (or scripted) languages were compared against the requirements. Scripted languages were seen as the most appropriate for autocoding applications, based on the information found during the literature review (Section 2.2).

Table 5 shows the 33 interpreted languages that were selected. Some additional languages, not shown in the table were removed for a number of reasons, which included: succession by a more recent version; very limited use and information; part compilation required; added a small amount of additional functionality to existing languages which were not required for this application.

Ant	APL	AppleScript	AutoIt	awk	BASIC	BeanShell
Ch	ColdFusion	Databus	ECMAScript	Flacon	Frink	F-Script
Game Maker Language	J	JASS	Lua	M	MAXScript	MEL
Mondrian	Perl	PHP	Pikt	PostScript	Python	Revolution
Ruby	Tcl	thinBasic	VBScript	PowerShell		

Table 5 - Scripted and interpreted languages for the next stage of research

4.1.3. Mandatory Requirements Filtering

The next stage of the language research was to apply the mandatory requirement filter to the list of scripted and interpreted languages identified in the previous stage (Table 5). The purpose of this filtering was to reduce the list of interpreted and scripted languages to a list of fundamentally appropriate and capable languages for the purposes of the autocoding process.

A spreadsheet was used for recording each language's fulfilment of the defined mandatory requirements. This spreadsheet, an example of which is shown in Figure 12 listed each language along with notes and each requirement's pass status. The full spreadsheet can be found in Appendix B.

Languages	Notes	Requirement Reference																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Ant	Ant was designed for the software build process. It is XML based and dedicated for this task, meaning that it is not a general purpose language.	✓	✗																
APL	APL is an array-programming language, and lacks any more general programming syntax and capabilities.	✓	✗																

Figure 12 - Sample from the mandatory requirements recording spreadsheet

Of the 33 languages identified during the previous stage only 5 passed the mandatory requirements, with the remaining 28 (~85%) failing. The 5 languages that remained after the mandatory filtering performed during this stage of the research were, in ascending alphabetical order: **Perl**, **PHP**, **Python**, **Ruby**, and **Tcl**. The fact that only 5 languages remained demonstrated that the research methodology has successfully narrowed down the languages to a manageable list of 5 languages that could be tested further in the next stage of the language research methodology.

4.1.4. Language Testing

Each of the 5 remaining languages from the previous stage was then tested in an autocoding application. The aim of the autocoding application test was to gain practical experience of the 5 languages, and help identify the advantages, disadvantages, current and possible future issues with each. The tests also verified that the satisfaction of the mandatory requirements researched previously were correct.

The test application covered the following fundamental aspects of an autocoding

application in the context of the projects undertaken during the NetGen analysis:

- Generation of multiple files using code templates;
- XML parsing;
- Input file validation;
- Mathematical computation;
- The use of multidimensional arrays;
- The use of cross template variables;
- The feeding of progress and status information back to the calling application;
- The checking of file and directory existence, and the creation of files and directories where required.

The automatic code generation application generated a C module and corresponding header file that provided functions for implementing the *Hill Cipher*. These functions allowed the user's application to encrypt and decrypt text in blocks of 3 characters. The user provided an XML formatted input file which contained the supported symbols for encryption, their corresponding codes, and the cipher's encryption matrix. From this the code generation implementation needed to validate the input file's contents, calculate the inverse matrix used for decryption, and then generate the C and header files.

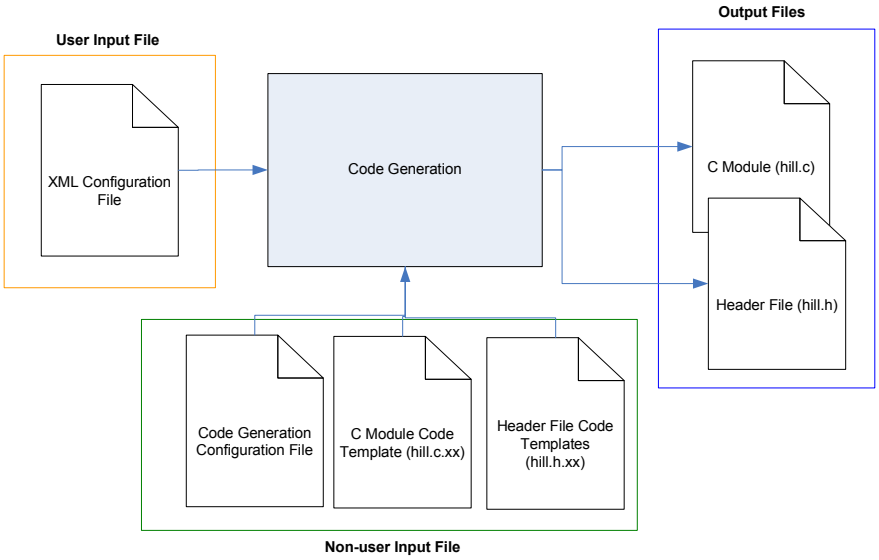

Figure 13 - Test code generation application input and output files.

The code generation implementation for each language consisted of 4 inputs files and 2 output files, shown in Figure 13. Further details of these tests can be found in Submission 4 [36]. A table summarising the test findings for each language can be found in Appendix C.

4.1.5. Optional Requirements Scoring

The optional requirements were used as an additional means of comparing the remaining languages against one another. The requirements consisted of both 'yes' or 'no' and multiple choice answers. To make the process more quantitative, a scoring scheme was used to provide a more meaningful indication as to what extent each optional requirement was satisfied. Each requirement was given a weighting based on Rapicore's interpretation of their relative importance. These weightings are shown in Table 6.

Requirement	Weighting
C-Like Programming Style	**1.0**
Literature & Community Support	**0.9**
Distributable Size	**0.8**
Readability	**0.7**
Dynamic Typing	**0.6**

Table 6 - Optional requirement weightings

A scoring scheme was defined for each requirement, and was chosen to suit the nature of the requirement in question. For example, the "Distributable Size" requirement's score was found by installing each language and recording the final installation size. A score out of 10 was then calculated as a percentage of the maximum value recorded from all the languages. After this the weighted of 0.8 was applied to get the final score for the distributable size requirement.

The optional requirement scoring process resulted in an individual requirement score and total score for each language. These scores were then used to aid the decision making process. A table showing the requirement scores can be found in Appendix D, with the total scores shown in Figure 14.

4.1.6. Language Selection

Up until this point the research methodology had performed the following:

- narrowed down the language population;
- reduced the remaining languages to those that are appropriate;
- tested each remaining language using a typical code generation application,
- scored each language against the optional requirements.

The research was then in a position to select the language to be used for developing the prototype autocoding platform.

During testing it was found that each language was very similar; perhaps due to the mandatory requirements used. However, each language had its own benefits and issues. Undoubtedly, however, from all the languages researched and tested the language that was most appropriate for the new autocoding platform was PHP. Although this language was perhaps the least likely to be the most suitable due to its typical implementation in server-side web development, PHP satisfied all mandatory requirements and scored top or joint top in all 5 optional requirements.

A cumulative graph displaying the optional requirement scores is shown in Figure 14, followed by a discussion of PHP's scoring for each.

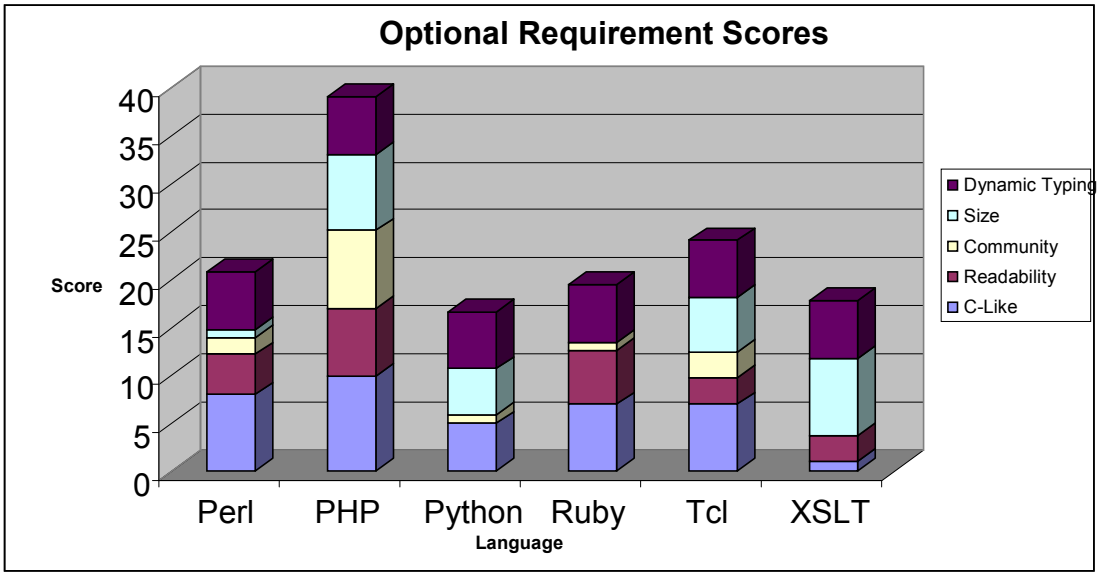

Figure 14 - Cumulative optional requirement score graph

4.1.6.1. C-like Programming Style

C-like programming style referred to the language's closeness in terms of syntax and semantics of the C programming language. This was chosen as a requirement due to the platform's future use for developing embedded applications; most of which are written in C. Selecting a language that was closer to C would make the development of the platform and the future code templates easier and quicker.

PHP was the most C-like of the languages tested. Most of the syntax is identical to that of C, and the basic programming constructs such as loops and conditionals were the same. One of the most useful similarities found during the test was that the file inclusion mechanism was the same as that used in C, i.e. the file is included as if it had been written where it was included. Other languages did not include the file, but instead executed the file. The inline inclusion makes understanding the behaviour and use of inclusions easier.

4.1.6.2. Readability

PHP is a flexible language, like C; allowing the programmer to freely format the code to aid readability without affecting the codes execution. The other 4 languages tested had some limitations on the positions of brackets or statements which prevented the code from executing. For example, the execution of Python is determined by the indentation of the code's statements. This makes it easy to accidently change the program's logic through careless indentations.

PHP was the only language to provide multiline comments, which can help the formatting and readability of the code. Of course multiline comments could be achieve in other languages by using multiple single line comments, but the effectiveness and readability of this method is decreased.

PHP also provides little "syntactic sugar", i.e. single symbols that perform the same operations as a number of longer statements, which means that difficult to read code is avoided. This syntactic sugar can be powerful and in many cases useful, but the whole practice of its use is often unnecessary. The same functions can be performed over a few extra lines and lead to far improved readability.

4.1.6.3. Community

PHP had the largest community and collection of literature. This is undoubtedly due to PHP's popularity and widespread use in dynamic websites. There are hundreds of books, and many tutorials and forums present to help any developer find the answer they need, or discover better ways of doing things.

PHP also had the most comprehensive and complete documentation of the languages tested. Not only were the functions fully explained in detail; the website also had an area on each page where users could contribute their own PHP experiences, tips, and code.

4.1.6.4. Installation Size

The distribution method most used by Rapicore for NetGen at present is via download from the company's website. This makes it more convenient for both Rapicore (for uploading) and the customer (for downloading) if the installation executable is as small as possible.

PHP had the smallest installation size of all 5 languages tested, coming in at only 6.31MB which was almost 30MB smaller than the next smallest installation (Tcl). This small installation also included all of the basic packages/modules required for the autocoding test performed such as mathematics, file processing, and XML.

Additional packages could be installed using the PHP Extension and Application Repository (PEAR), or during the installation. The additional list of modules is large, and includes functions such as database access, encryption, and compression. All of these could potentially be used during more advanced autocoding applications.

4.2. Autocoding Platform Development

Using PHP that was selected using the methodology described in the previous subsection, the prototype autocoding platform was then developed using industry recognised best practices, all of which were based on the development model selected.

These best practices included: a stakeholder analysis; functional and non-functional requirements capture; component design using UML; implementation using a suitable coding standard; unit, integration, and user acceptance testing.

PHP was utilised by the platform for two aspects of the platform: 1) to develop the platform itself, i.e. front-end, validation, code generation, etc; 2) the implementation of dynamic processing in the code templates (Section 6.2.2). This subsection will summarise the development of the prototype autocoding platform, covering the development methodology and the stages performed during the development. A detailed explanation of the development can be found in portfolio Submission 5.

4.2.1. Development Model

Submission 1 discussed a number of commonly used software development models which included the V, Waterfall, Spiral, and Agile-based models. Each of these models has their own advantages and disadvantages, with some being suited to particular projects more than others.

It was decided to use the V-Model for the development of the autocoding platform. The V-Model was chosen because it:

- is simpler and easier to follow than the spiral and agile-based models;
- provides a clear top-down development, bottom-up validation methodology;
- helps prevent bugs earlier in the development, saving development time (as well as cost in other projects);
- provides direct links between the design and testing stages;
- is suitable for single developer projects;
- allows the developer to customise the development stages.

There are many different variations of the V-Model, where developers and organisations have tailored the model to suit the nature and needs of the projects they undertake. The tailoring of the model usually involves the removal or renaming of the stages from the 'base' model. A diagram of the V-Model variant used for the autocoding platform's development is shown in Figure 15 below.

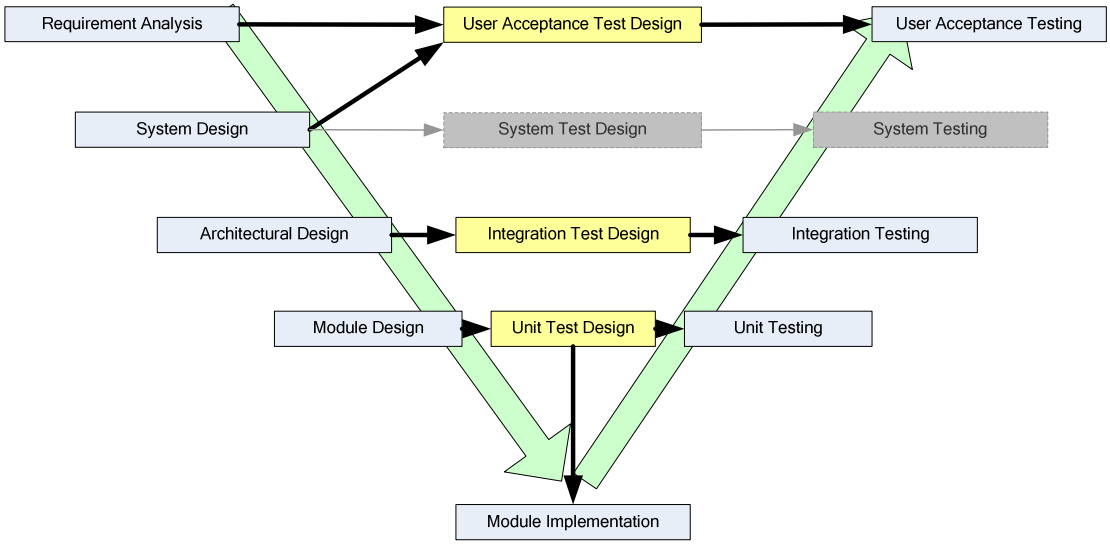

Figure 15 - The V-Model variant used for the autocoding platform's development

All of the stages shown in Figure 15, including those in grey, form the base V-Model. For this development the system test design and system testing stages (in grey) were removed from the base and combined with the User Acceptance Testing (UAT) stages to form the variant used. Since this development was a research-based prototype with only a single developer involved in the testing stages, the implementation of both UAT and system testing stages would have involved a lot of test repetition. It was therefore more appropriate to combine the two testing stages into single UAT design and testing activities. This arrangement better suited the prototyping and single developer characteristics of this development.

In hindsight, although the principle of combing the UATs and system tests was sound, the tests should have been referred to "System Tests" rather than "UATs". Referring to the tests as "UATs" (regardless of whether they are the same tests or not) is misleading since the tests were not going to be performed by a 'user' of the system; only the developer.

4.2.2. Requirements Analysis

The requirements analysis stage consisted of three activities: *Stakeholder Analysis*, where the stakeholder's in the platform and their needs were identified; *Requirements Definition*, where the requirements of the platform were defined; *User Acceptance Test design*.

UN	Technical Users		
	These are the requirements of the users with technical coding and code generation experience.		
Reference	Requirement/Category	Importance	Related Requirements
UN.1	Functional		
UN.1.1	Options		
UN.1.1.1	The platform must allow the user to view the different languages available	Mandatory	
UN.1.1.2	The platform must allow the user to select the language required for code generation	Mandatory	
UN.1.1.3	The platform must allow the user to view the different compilers available	Mandatory	
UN.1.1.4	The platform must allow the user to select the compiler required for code generation	Mandatory	
UN.1.1.5	The platform must allow the user to view the different targets available	Mandatory	
UN.1.1.6	The platform must allow the user to select the target required for code generation	Mandatory	
UN.1.2	Formatting		
UN.1.2.1	The user must be able to edit the formatting used	Mandatory	
UN.1.2.2	The user must be able to save and load the formatting used	Mandatory	
UN.1.2.3	A default formatting must be available	Mandatory	
UN.1.2.4	The platform must support a range of commonly used formattings	Mandatory	
UN.1.3	Name Styling		
UN.1.3.1	The user must be able to edit the name styling used	Mandatory	
UN.1.3.2	The user must be able to save and load the styling used	Mandatory	
UN.1.3.3	A default styling must be available	Mandatory	
UN.1.3.4	The platform must support a range of commonly used stylings	Mandatory	

Table 7 - Example of Technical User requirements capture spreadsheet

The prototype autocoding platform's requirements were recorded in a spreadsheet, using a technique representative of commercially available requirements capture tools,

e.g. IBM Rational DOORS [37]. This contained the defined functional and non-functional requirements under the various stakeholder categories, along with a reference number that could be used to trace the requirement's fulfilment and an importance rating (Mandatory, Desirable, and Luxury). An example of these requirements is shown in Table 7, with a full list of the defined requirements available in Appendix B of Submission 5 [38].

A total of 90 requirements were defined for the platform. These requirements were used for the system, architectural, and module design stages of the development discussed in the following three subsections.

4.2.3. System Design

During the system design stage, initial concepts, and the techniques and methods required to meet the platform's requirements were identified. Figure 16 shows an abstract system representation of the platform and the various inputs and outputs to and from the platform.

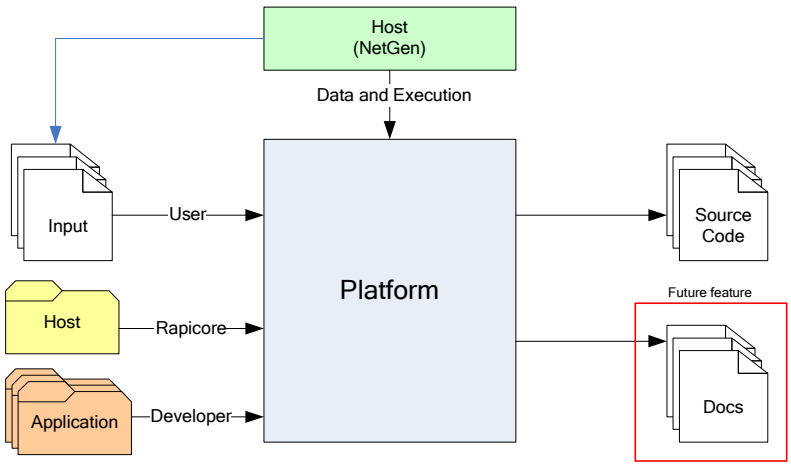

Figure 16 - Abstract platform diagram of system inputs and outputs

At the system level, the platform consisted of three basic input types: *User*, *Rapicore*, and *Developer*. The user inputs, in addition to their customisation options (language, compiler, target, etc) were the input data files for the code generation process. As the platform was developed in the context of NetGen, these input data files took the form of System Description Files (SDF) created by NetGen.

The Rapicore input was the host information which allowed the user to configure the host related settings. Finally, the developer inputs were the various code generation applications which the user could select between. Each application represented one particular collection of source code outputs, for example, one application may generate a J1939 communication stack and another may generate a FlexRay communication stack.

Some of the key concepts decided upon during the system design stage and later implemented in the platform included:

Graphical User Interface

The platform provided a graphical user interface (GUI), partly based on the literature review findings in Section 2.2.2, and was developed using the open source GTK+ toolkit, which is a collection of libraries for developing GUIs [39]. This user interface allowed the user to configure and control the autocoding process. Being written in PHP, it also allowed the core processing modules to easily interact with the user interface and the user. This would otherwise be difficult if the GUI was written in a compiled language.

Host Flexibility

The platform allowed Rapicore to interchange a GUI component dedicated to the host and its configuration options. This would allow Rapicore to use the autocoding platform with development tools other than NetGen should the need arise in the future; adding to the platform's flexibility.

Multiple Generations

The platform allows the user to automatically generate source code for multiple targets; each being known as a 'generation'. In addition to this, the user can also configure the language, compiler, and target options of each generation independently.

Information Display

The platform uses a display component within the main window to display the current state and progress of the autocoding process, as well as presenting any errors and warnings to the user when required. This feature was not available with the XSLT implementation.

Multiple Applications

The platform was designed around an application concept, where each application generates a collection of source code files for one particular purpose, as mentioned previously. The user can select between their installed autocoding applications, configure any application specific options through a dedicated application configuration GUI component, and then generate the source code for that application.

Application XML

One of the main concepts implemented in the platform is that of Application XML files. These files contain all of the application data and its associated code templates used during the autocoding process. Two application types exist (Basic and Advanced), both of which were specifically designed to support the various requirements of the platform. Each combines the flexibility and expandability of XML, with the computational and mathematical power of PHP to support the dynamic autocoding requirements. This autocoding method will be described specifically in this report's discussion, as it relates to the innovation claimed for this project.

4.2.4. Architectural Design

The architectural design stage focused on the main abstract modules that were needed to implement the platform. These modules and their hierarchical positions within the platform are shown in Figure 17.

The autocoding platform was divided into two groups of components: the *GUI* and *Core*. The GUI was used for interacting with the platform and for configuring the code generation process. The Core received 'commands' from the GUI and performed the main processing within the platform; primarily that of code generation.

The Core's main processing responsibilities include validation, code generation, and process control. The core is made from a number of other processing elements, such as the Generator which is responsible for controlling and generating the different outputs, i.e. source code and documentation. At present only the code generation (based on the

defined requirements) was considered, but documentation could be supported in the future through this module.

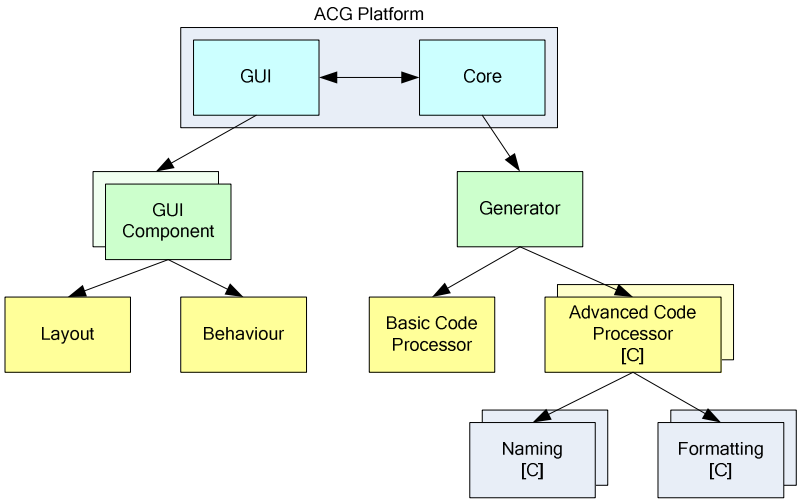

Figure 17 - The Autocoding Platform's high-level (architectural) design

For the code generation process the generator used one of two code processors: the basic code processor and the advanced code processor (corresponding to the basic and advanced application XML implementations). The basic code processor could be used by the developer for implementing quick and/or simple code generation applications.

The advanced processor can be used for generating fully customised code based on the language, compiler and target selection. There was one advanced code processor for each language, although only the C language had been implemented for this project. The advanced code processor made use of two further modules: the *Formatting* processor and the *Naming* processor. Again, there would be a set of these for each language. As the name suggests, the formatting element was responsible for the generation of formatted code, and the naming processor was responsible for the naming of identifiers within the formatted code.

4.2.5. Module Design

The module design stage of the prototype autocoding platform's development transferred the deliverables from the architectural design stage into a design for each of the individual modules that made up the platform. This design stage primarily used UML and an associated design tool called IBM Rational Tau [40]. An example of one of the UML diagrams, this one being for the Advanced Code Processor is shown in Figure 18.

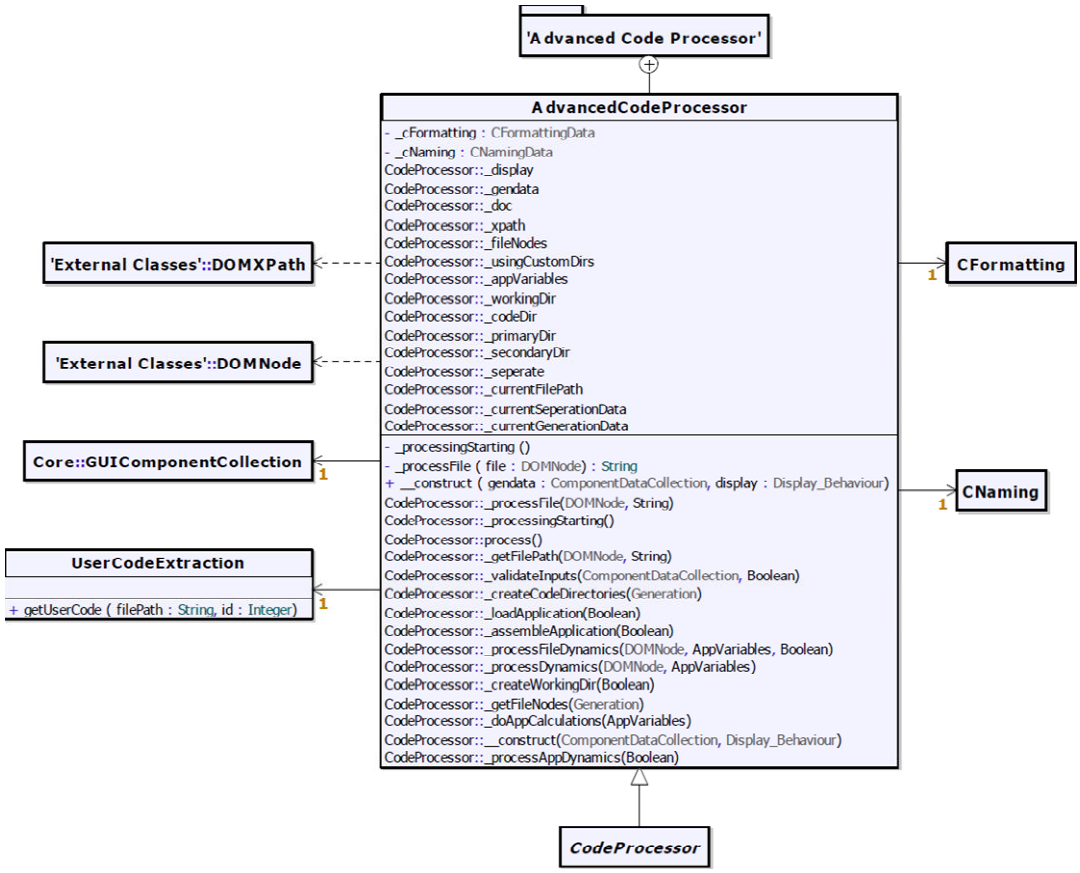

Figure 18 - An example UML class diagram for the Advanced Code Processor

Each software module was designed using the Zend Framework's naming conventions [41]. This specified the naming conventions used for the likes of: public and private class variables, classes, and the file naming and content conventions. For example, private class variables begin with an underscore and are written in Camel-Case, e.g.

`_myVariable`. In addition to the Zend naming conventions, the platform also implemented a number of autocoding platform specific naming conventions for GUI components and their associated filenames.

In addition to the behavioural and structural design, the GUI component layouts were also designed.

4.2.6. Implementation

Using the module designs, the GUI and core components of the prototype autocoding platform were successfully implemented using: GTK+, which was described previously; Glade, which is a GUI designer for GTK+; and the PHP scripting language. The software metrics for the developed autocoding platform are summarised in Table 8.

Total lines	12’870
Total lines (excluding XML based files)	10’082
Total size	656’565 bytes
Total size (excluding automated files)	450’571 bytes
Total public attributes	155
Total private/protected attributes	96
Total public methods (exc. Interfaces)	144
Total private/protected methods	120
Total functions	45
Files	53
Classes	33
Interfaces	1

Table 8 - Autocoding platform software metrics summary

This subsection will overview the main GUI components developed during the project. A more detailed explanation of the implementation can be found in Submission 5 and its accompanying CD.

4.2.6.1. Main Window, Host, Information, and Command Components

The Main Window, Host, Information Display, and Command components developed are shown in Figure 19. This screenshot is what the user sees once the platform has successfully loaded and initialised the required components.

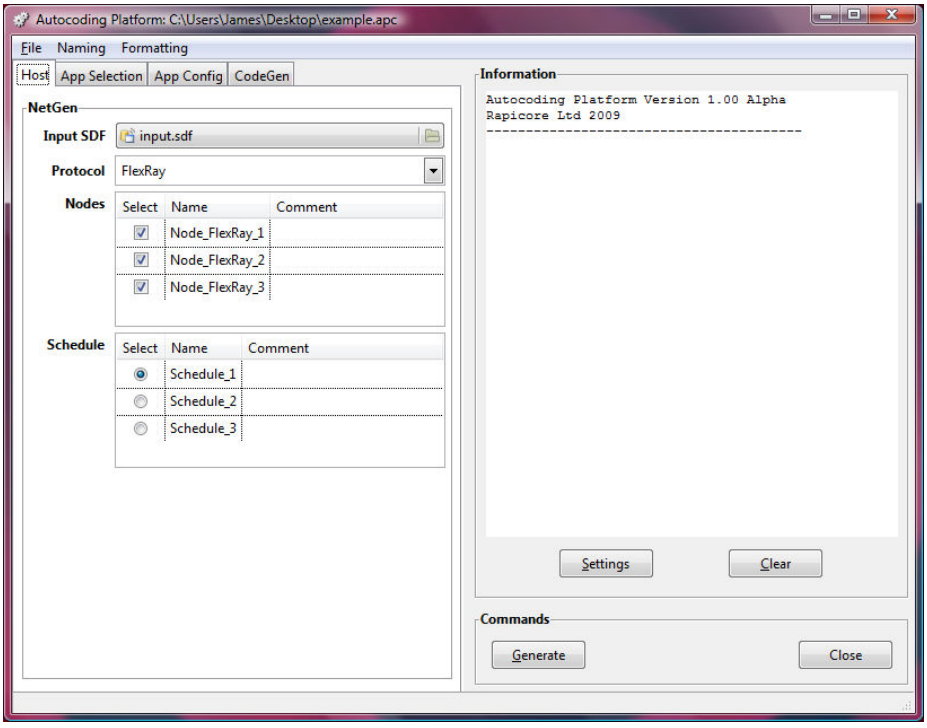

Figure 19 - Main Window, Host, Information, and Command components

The implementation of each component visible in Figure 19 is summarised below.

Main Window Component

The main window (Figure 19) is responsible for: holding the majority of the GUI components; displaying the formatting and naming components; saving and loading the

user's configurations. This is the first component loaded by the Core so that the required components can be placed inside.

Using the GTK+ toolkit it was possible to develop a main window component that was fully resizable, meaning that the user could expand and reduce the window as required to suit their screen. In addition to resizing the main window, the left and right areas of the GUI can be resized using the invisible divider or boundary in the middle of the window. This allows the user to increase the size of the information display when the settings on the left are not currently required and vice versa.

Host Component

The host component (Figure 19: Left) allows the user to configure host specific options relating to the code generation process. This will usually include the location of input file(s), as well as a number of additional host specific options. The platform allows Rapicore to develop separate host GUI components so that the available options can be tailored for hosts other than NetGen.

The implementation of the host component was straight forward. The selection and de-selection of the network nodes had close interactions with the Code Generation component's generation list. This functionality was fully tested during the integration testing stage of the development.

Information Component

The information display (Figure 19: Top right) is responsible for controlling and displaying the information from other components to the user. This information covers general information, warnings, errors, processing status, and the state of the platform

itself. Any GUI and non-GUI component can display information on the information display. The display is also responsible for creating log files which can be later used by a developer for debugging the platform.

Commands Component

The commands component (Figure 19: Bottom right) is responsible for sending commands to the core and other GUI components. For the prototype, the command component is used to begin the generation process and to close the platform when requested by the user. The future intention of this component is to provide a full command line interface for the user to interact with the platform.

4.2.6.2. Application Selection Component

The implementation of the application selection GUI component is shown in Figure 20. The application selection component is responsible for: gathering the available applications; allowing the user to select their required application; displaying information on the currently selected application. The application's information includes: description, version, copyright, and the website of the developer.

The Application Selection component currently displays the most important information regarding the selected application. There is, however, plenty of free space in the component to display any additional application information in the future if required.

The Application Selection component interacts with the Application Configuration component by loading the application configuration GUI component (if one is present) for the application selected. This interaction was successfully tested during the integration testing stage of the project.

Figure 20 - Screenshot of Application Selection component's implementation

4.2.6.3. Code Generation Component

The implementation of the Code Generation component is shown in Figure 21. The code generation component allows the user to select one or more generations and the options required for each generation. The platform allows the user to generate multiple versions of the selected application with a single click of the button. The user can add or remove a generation, and can also select different code generation options (folder, language, compiler, and target) for each generation added. The user also has options common to all generations which include the selection of the code's main output directory and whether the source files are automatically separated or not.

At present the supported languages are implemented manually in each of the required components (File Separations for this component). In hindsight, it would have been better to implement some form of language support method so that languages can be

easily added and removed without modifying the code of each component. In terms of generic learning, whenever a software application needs to support the addition of related components (language support components in this instance), a single, centralised module should be implemented. Other components would then be able to query this same component through a common interface to retrieve information about the components currently installed/supported. This method improves expandability, reduces the risk of coding errors, and removes the need to repeat the unit testing of existing components whenever a new component is added.

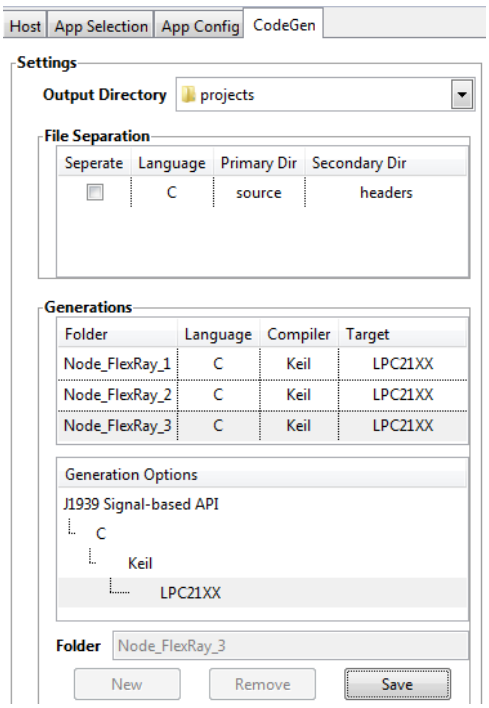

Figure 21 - Screenshot of Code Generation component's implementation

The generation list allows the user to set the 4 main options for each generation, namely the folder, language, compiler and target. In the future this list could also allow the user to set additional options such as code optimisation or coding standard support if

required. Due to the requirements set for the platform, the additions of these features at present were not seen as necessary.

4.2.6.4. Formatting Component

The formatting component allows the user to modify the formatting of the source code generated. The user can select between a range of different items that can be tailored, such as the formatting of loops, conditionals, expressions, and functions. These can be modified and saved by the user, and then used by the core to generate their desired code.

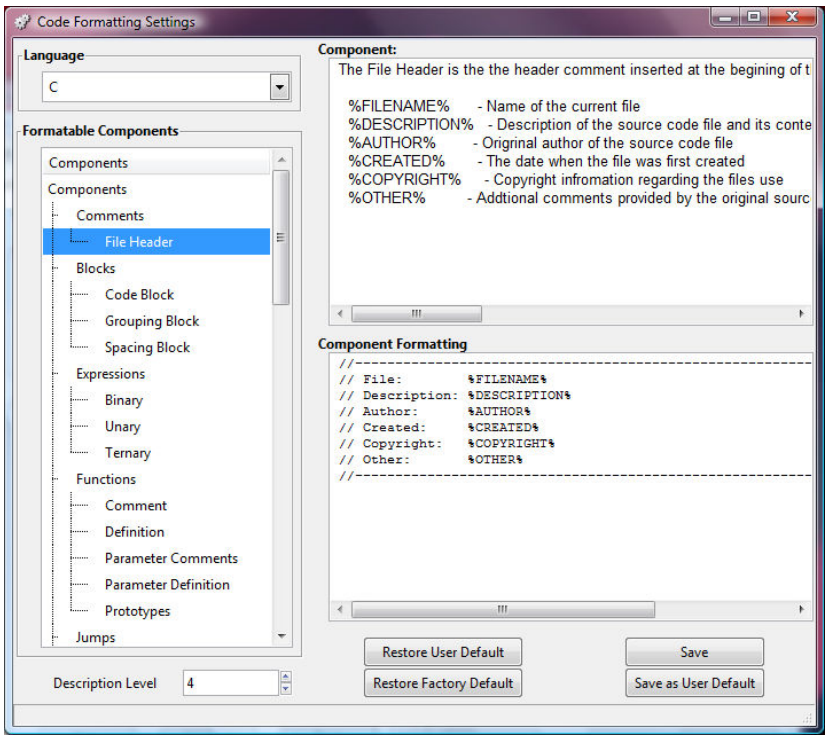

Figure 22 - Screenshot of formatting component's implementation

The formatting component (Figure 22) uses an XML file for loading and storing the user's formatting settings. This file is called `LanguageFormats.xml`, and is contained in the formatting GUI component's directory. It contains the following for each formattable component in the platform: the category of the item; name of the item;

the standard (‘factory’) default formatting; user default formatting; current formatting; description of the formatting item. The file contains these for each language supported by the platform. This component uses this file and the user’s language selection to retrieve the data and display it in the available widgets.

The one feature not currently implemented in the prototype that would have been useful, and that is available in the Naming component is that of macro functions. These macro functions allow the user to automatically insert formatting mark-ups to the current settings without having to write them manually.

An additional feature was added to the formatting component window that was not originally designed for. This feature (Figure 22: Bottom left, “Description Level”) allows the user to select the level of the descriptions output to the final source code files. The application developer assigns an ‘importance level’ to each description in the application XML file. Important descriptions are given a low level number and less important, i.e. obvious, descriptions are given a high number. The user can therefore select the maximum level they require in the output. They may also choose a value of ‘0’ which omits all descriptions from the output.

In hindsight, this feature should have either been included in the original requirements or the feature should not have been added so late in the platform’s development. However, since the development was of a prototype it was seen as more beneficial to test out the theory, especially considering the fact that the effort required to add this feature was trivial and required very little change to the design of the existing components.

4.2.6.5. Identifier Naming Component

The naming component allows the user to modify the naming of identifiers present in the output source code. As with the formatting component, the user can modify and save the desired naming styles so the code output meets their requirements.

The implemented identifier naming component is shown in Figure 23. The operation of the naming component is similar to the formatting component shown previously. The component uses an XML file for loading and storing the user's naming settings. This file is called `LanguageNaming.xml` and is contained in the naming GUI component's directory. The contents of this file are similar to those in the formatting component's file.

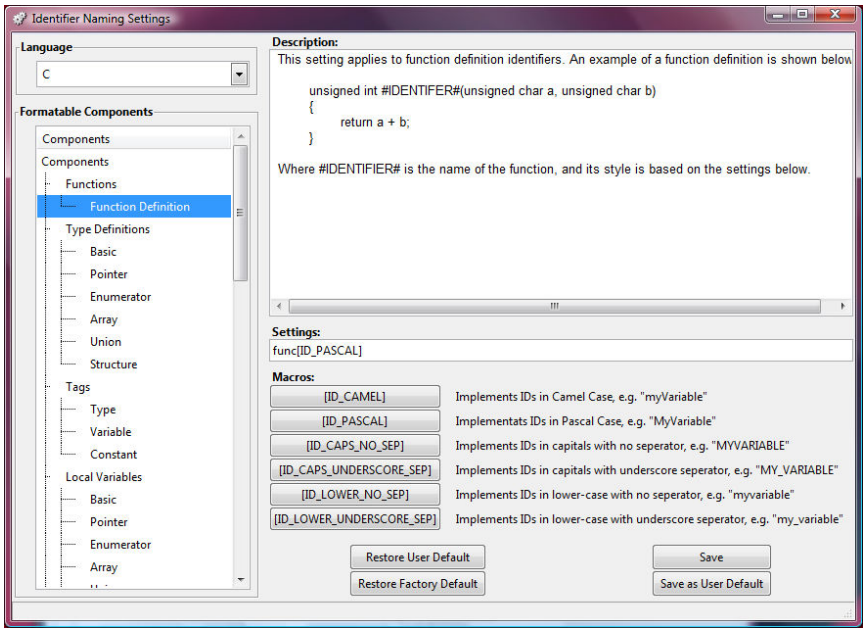

Figure 23 - Implemented Identifier Naming window

4.2.7. Validation

The validation stages of the autocoding platform development project consisted of 3 groups of tests: *Unit Tests*, *Integration Tests*, and *User Acceptance Tests*.

A range of tests with individual test cases were defined and developed into sets of testing logs. These logs contained: testing numbers; test descriptions; the actions required; and the expected results of each test and test case. Each test was performed sequentially in a defined order, and the actual results were compared to the expected results. If an error was identified then the module/s in question were rectified and the tests were re-run to ensure that the error was corrected. If this rectification was successful then the log was marked as such and the testing was resumed.

The following subsections summarise the groups of tests performed. All of the testing documents, as well as the PHP test files (where available) are provided on the CD that accompanies Submission 5.

4.2.7.1. Unit Testing Summary

A summary of the modules tested, the number of tests performed, and the issues identified are provided in Table 9.

Module	Tests (Test Cases)	Passes (Failure)	Issues Identified
Main Window	7 (18)	18 (0)	-
Host	7 (20)	19 (1)	Filename changing, but the nodes and schedules were not being loaded from the configuration file.
Application Selection	5 (12)	12 (0)	-
Code Generation	9 (25)	24 (1)	Invalid directory returned when a valid directory name was entered
Formatting	5 (12)	12 (0)	-
Naming	5 (12)	11 (1)	Saved settings not displayed when the platform is closed and reopened.
Information Display	4 (7)	7 (0)	-
AppVariables	1 (10)	10 (0)	-
Generator	1 (5)	5 (0)	-
BasicCodeProcessor	1 (20)	20 (0)	-
UserCodeExtraction	1 (6)	6 (0)	-
Total	**46 (147)**	**144 (3)**	

Table 9 - Module Test Summary

The majority of the modules designed could be implemented using the module tests. The one exception was the `Core` class because two of its main functions were to load and initialise the GUI, and to gather the data from the required components. This of course could not be done without the existence of the GUI components. This test was therefore carried out as an integration test.

4.2.7.2. Integration Testing Summary

The sets of tests performed during the integration testing stage of the development are summarised in Table 10 (in chronological order). The integration tests were initially developed during the architectural design stage. It was necessary to adjust these tests during the module design to ensure that each test matched the low-level implementations of the components. This need for adjustment demonstrated the difficulty in writing high-level tests so far ahead of the low-level design and implementation. It also demonstrates the issues associated with the V-model development methodology and the benefits of more iterative or agile alternatives.

Module	Tests	Passes (Failures)	Issues Identified
Core – GUI Components	6	5 (1)	Error in MSDOS. GUI fatal error and closes (unable to prevent)
Host – Code Generation	7	7 (0)	-
Application Selection – Application Configuration	8	6 (2)	Applications present when they should not be
Application Selection – Code Generation	2	2 (0)	-
Main Window - Formatting	3	3 (0)	-
Main Window – Naming	3	3 (0)	-
Core – Generator – BasicCodeProcessor	1	1 (0)	-
Core – Generator – AdvancedCodeProcessor – CNaming - CFormatting	6	6 (0)	-
Total	**36**	**36 (3)**	

Table 10 - Integration Test Summary

The tests were primarily based on modules that interacted with one another, and were

arranged so that the full platform was gradually constructed and tested one module at a time.

4.2.7.3. User Acceptance Testing Summary

A summary of the tests implemented is shown in Table 11. The user acceptance tests were very similar to the individual module tests. Unlike the unit tests, however, these aimed to test the fully integrated functionality of the platform as a whole. The UAT would usually be carried out by a user of the system, however, for this project the UAT and System Tests were combined to form a single set of tests to avoid repetition.

Test Description	Test Cases	Passes (Failures)	Issues Identified
Initialisation	1	1 (0)	-
User configuration saving	3	3 (0)	-
Creating new configuration	3	3 (0)	-
User configuration loading	2	2 (0)	-
Automatic configuration loading on start-up	3	3 (0)	-
NetGen host and generation list control	5	5 (0)	-
Application selection and configuration loading	2	2 (0)	-
Naming configuration	5	5 (0)	-
Formatting configuration	6	6 (0)	-
Code generation configuration	2	2 (0)	-
Code generation process	8	8 (0)	-
Logging	2	2 (0)	-
Total	**42**	**42 (0)**	

Table 11 - User Acceptance Tests Summary

Due to the unit and integration tests performed, all of the UAT tests were carried out successfully.

4.2.7.4. Requirement Fulfilment Summary

A total of 90 requirements were defined for the autocoding platform based on the stakeholder's identified and their expectations and interests. This number consisted of 67 mandatory, 20 desired, and 3 luxury requirements. Table 12 below quantitatively

summarises the number of requirements defined and fulfilled for each of the stakeholders.

	Mandatory		Desired		Luxury		Total		
Stakeholder	**# Defined**	**# Fulfilled**	**# Defined**	**# Fulfilled**	**# Defined**	**# Fulfilled**	**# Defined**	**# Fulfilled**	**# Not Fulfilled**
Academic	5	5	0	0	0	0	5	5	0
All Users	12	11	5	5	2	0	19	16	3
Development Teams	1	1	1	0	0	0	2	1	1
Non-Technical Users	4	4	0	0	0	0	4	4	0
Technical Users	25	25	0	0	0	0	25	25	0
Application Developers	9	7	5	1	0	0	14	8	6
Platform Maintainers	4	3	0	0	0	0	4	3	1
Directors/Rapicore	4	4	4	2	1	1	9	7	2
Marketing/Distributor	3	2	5	4	0	0	8	6	2
Total	**67**	**62**	**20**	**12**	**3**	**1**	**90**	**75**	**15**

Table 12 - Requirement fulfilment summary by stakeholder and requirement type

Focusing on the mandatory requirements, of the 67 requirements the design and implementation of the platform has managed to fulfil over 92% of those defined. Of the 5 requirements that were not fulfilled (Table 13), 3 of them related to the provision of documentation for the platform maintainers, autocoding application developers, and users of the tool. In hindsight, these requirements should have been defined as luxury for the development of a prototype, and only be defined as mandatory for a version intended for retail.

Reference	Requirement
AU.1.3.3	The platform must check that the configurations are not corrupted before they are loaded.
AD.2.1	Complete documentation should be provided on how to create applications
AD.2.2	Documentation of all parts of the platform required for application development
PM.2.1	Documentation detailing the full design and operation of the platform must be available to the maintainers
M.2.6	Installer for distribution if retailed separately from NetGen

Table 13 - Mandatory requirements not fulfilled by the platform

Requirements AU 1.3.3 and M 2.6 should have been defined as desired or luxury. Referring to requirement AU 1.3.3, although it is important that corrupt configurations are not loaded without the corruption being detected, for a prototype development the

importance of such a requirement is lessened and therefore takes a lower priority over other requirements.

The final requirement (M 2.6) should not have been defined as a requirement at all (and especially as a mandatory requirement) as it contains a conditional clause, i.e. “if retailed separately”. This highlights a mistake in the requirements capture process and demonstrates the importance of a more formal requirement peer review process.

5. Prototype Platform Evaluation

The development of the PHP-based prototype autocoding platform, discussed in the previous section, was based on a number of recommendations which originated from the issues and limitations identified with NetGen's XSLT-based autocoding method. The fulfilment of these requirements through the design, implementation, and testing resulted in a fully functional prototype autocoding platform that implemented new autocoding methods to solve the issues identified.

Although the platform was designed, developed, and validated using requirements originating from the issues identified, the platform had yet to be evaluated against the recommendations proposed in Submission 3. In addition, the effectiveness and practicality of the platform's two code generation methods had not been tested using a real application.

The aim, therefore, was to perform a conclusive evaluation of the prototype autocoding platform and the autocoding methods implemented. This section will cover the defined criteria (5.1), a case study which aimed to evaluate the more practical criteria and to compare the two code generation methods employed (5.2.), and summarise the results of the evaluation (5.3.).

5.1. Evaluation Criteria

To evaluate the prototype autocoding platform a set of criteria were defined based on the issues identified and the recommendations presented in Submission 3 [29]. In addition to these criteria it was necessary to evaluate the platform and its employed code

generation methods to ensure that additional issues had not been created. For the new platform to be successful all of these criteria needed to be satisfied. These criteria and metrics are summarised in Table 14, with full descriptions of these criteria available in Submission 6 [42].

Number	Criterion
1	Mathematical functions and computational capabilities
2	Inherent functionality and constructs
3	Output formatting capabilities
4	Template granularity and reusability
5	Standalone Deployment
6	Output configurability
7	Ease of integration with NetGen
8	Cost effective
9	Development Time
10	Language, Compiler, and Target selection
11	Identifier naming and code formatting
12	Cross template variables
Number	**Application Metrics**
13	Implementation time
14	Execution time (for generating the code)
15	Application XML size
16	Ease of implementation

Table 14 - Prototype autocoding platform evaluation criteria

5.2. Case Study

A case study was used to implement the same autocoding application using both the basic and advanced code generation methods. The objectives of this case study was to evaluate the criteria that were only supported by the platform (and not yet tested), to compare the two code generation methods implemented against a number of metrics defined in the previous subsection, and to ensure that no issues had been created. The implementation of the same application using both code generation methods meant that a more accurate comparison of the two processing methods could be attained.

The first task was to decide upon an application that would effectively test both the basic and advanced applications. Ideally the case study chosen would have been the FlexRay project which failed to be completed successfully in Submission 3. The prototype could then have been used to show that the additional functionality and features provided by the platform would now allow the same project to be completed successfully. Unfortunately, however, the use of this application for the case study was not possible. This was due to the time period elapsed since the original FlexRay project (approximately 3 years). Most of the expertise and knowledge required for the project has been lost and the other project members have since moved on.

Instead, a new application was developed that covered the same aspects as those required to successfully complete the FlexRay project. These requirements included the following:

- Multiple output files;
- Input validation;
- One-time calculations;
- Sharing of data between code templates;
- Control over output formatting;
- Mathematical and computational ability.

The application chosen for the case study was a CAN message API and driver stack, an abstract diagram of which is shown in Figure 24. The message API provides the user's application with a number of functions for setting, retrieving, and transmitting messages on the CAN bus. The interface hides the CAN message information and data; allowing the user to manipulate the messages based on each message's name (defined in the SDF file using NetGen), rather than by each message's CAN identifier.

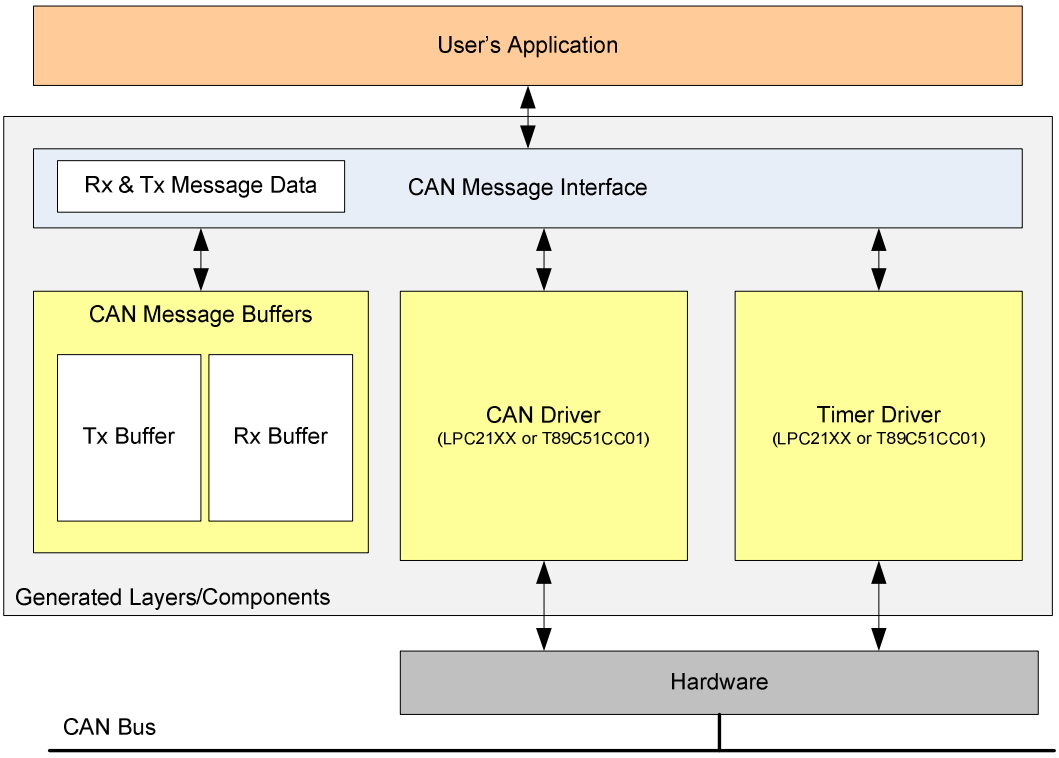

Figure 24 - Abstract diagram of the evaluation software stack

The features of the stack are listed below:

- Message-oriented transmission, reception, and data reading/writing;
- Message transmission and reception status;
- Variable transmission and reception buffer sizes;
- First-In-First-Out (FIFO) or First-In-Last-Out (FILO) buffer configuration;
- Independent configuration of transmission and reception message buffers;
- Configurable CAN bitrates (125, 250, 500, 1000 Kbit/s);
- Configurable buffer service period (in milliseconds);
- Support for 3 compilers and 2 targets:
    - LPC21XX (RealView and GCC compilers);
    - T89C51CC01 (Keil C51 compiler).

5.3. Evaluation Results

Through the rigorous and systematic selection of the PHP language for the autocoding platform, and the set of requirements against which the various languages were

compared, all of the criteria described were already met to a greater or lesser extent. These criteria relate to topics such as: the mathematical and computational ability of the language, and the programming constructs available. In addition to these, some criteria were also satisfied through the requirements definition, design, implementation, and testing of the autocoding platform itself.

A summary of the evaluated criteria and whether they were satisfied is shown in Table 15. A discussion of the criteria and how each was satisfied by the platform and the autocoding method can be found in Submission 6 [42].

Number	Criterion	Satisfied
1	Mathematical functions and computational capabilities	**Y**
2	Inherent functionality and constructs	**Y**
3	Output formatting capabilities	**Y**
4	Template granularity and reusability	**Y**
5	Standalone Deployment	**Y**
6	Output configurability	**Y**
7	Ease of integration with NetGen	**Y**
8	Cost effective	**Y**
9	Development Time	**Y**
10	Language, Compiler, and Target selection	**Y**
11	Identifier naming and code formatting	**Y**
12	Cross template variables	**Y**

Table 15 - Criteria and criteria satisfaction summary

The results of the metrics for the basic and advanced application types are summarised in Table 16.

The aim of comparing the two application types was to evaluate whether the original intentions of having two types had been realised. The aim of having a basic application method was to allow Rapicore to develop autocoding in less time and with greater ease than the advanced applications; at the expense of the inability for the user to customise

the identifier naming and code formatting. As shown in Table 16, this was satisfied with a 4 day less implementation time and an easier rating for the development of the application.

Metric	Basic	Advanced
Implementation time (13)	~2 days	~6 days
Execution time (14)	23 ms	1076 ms
Application XML size (15)	95 Kbyte	174 Kbyte
Ease of implementation (16)	Easy/Medium	Medium/Difficult

Table 16 - Application metric comparisons

The two processing methods were found to be very effective in creating customised source code using the prototype autocoding platform. The original intentions of having the two code processing types was satisfied, with the advanced method taking longer to implement than the basic method, but with the benefits of providing code formatting and identifier naming customisation which was not possible with basic applications. It is also worth noting that, after the development of some additional tools to aid the advanced process, e.g. a tool for keeping track of variable references, the implementation time of advanced applications would be comparable to that of basic applications.

A number of issues were identified that related solely to the advanced code generation method. These issues were caused by a number of oversights during the formatting and naming processor's design and implementation. These did not affect the successful generation of the source code using this method, but they did require a temporary fix during the case study undertaken. It is recommended that some additional analysis and redesign is performed prior to using advanced applications immediately in a production environment (Section 8). These issues demonstrated the potential benefit of additional

designers during the development process. These designers would provide additional perspective, as well as reducing the risk of important needs and requirements being missed (as was the case here).

In terms of the application implementation time, execution time, and application XML file size, the advanced application's metrics are less than ideal. However, this is only relative to the basic application's metrics and in reality the measured values had little impact on the efficiency and effectiveness of this type of application.

The main aspect that would influence the choice between the two applications would be that of implementation difficulty, where the advanced application was more difficult to implement than the basic or the original XSLT method. However, it was known that with the provision of a set of tools to aid the developer for advanced applications, not only could this process be made easier; the time required to develop such an application could be reduced.

To conclude the evaluation, one could compare the implementation time and difficulty of the current autocoding method to the XSLT-based method. This is because the generated CAN API was similar to that created for the CAN project discussed in Section 3.1. The XSLT-based CAN API generation templates took approximately 5 days to implement. This is comparable to the time required for the advanced application's implementation. As for implementation difficulty, it would be rated as Medium; putting it somewhere between the basic and advanced methods. Much of the time and difficulty with the old method was associated with the limitations (mathematical, etc) of the XSLT language.

	Autocoding Method	
	XSLT	**XML/PHP**
Pros	• Relatively simple implementation • Well documented language • Easily integrated with NetGen	• Two methods (simple and advanced) to suit the development requirements • Full mathematical and processing functionality with PHP • More structured template implementation with XML • More code generation options to tailor the code generated • Code styling and formatting • Code template portability due to use of XML • Generic language (PHP) providing all of the functionality of a regular language
Cons	• Lack of mathematical functionality • Difficult to format output • Lack of basic constructs and computational functionality • Lack of variables and data types • Cross-template variables not available • Not a generic language – used for transforming XML documents	• Advanced applications are time consuming to develop with tools to aid the process. • XML and PHP are well tested, but the combination of the two has little peer testing.

Table 17 - Autocoding method comparison table of pros and cons

Table 17 provides a benchmark between the original XSLT-based autocoding method and the XML/PHP based autocoding method implemented in the prototype autocoding platform. As one can see from the table, the pros of the new method far outweigh those of the XSLT-based method, and the cons can be solved with some additional development effort and testing (see Section 8: Future Work).

6. Discussion

Sections 2 to 5 discussed the four stages of the research and development undertaken, as proposed by the research methodology (Section 1.2.2). This research and development resulted in a functional prototype autocoding platform which has been designed, implemented, tested, and evaluated based on the issues identified with the XSLT-based method used by NetGen.

From the beginning of this project, the area of innovation was to be associated with the methods and techniques used for the autocoding process. The main aim of this section is to discuss the proposed innovative autocoding method implemented. The section begins by defining the term ‘innovation’, and then stating the innovation claimed in this project (6.1). The section then describes the operation of the autocoding method (6.2). Once an understanding of the method has been gained, Section 6.3 then justifies the claim of innovation; based on the definition provided in Section 6.1. Section 6.4 moves away from the proposed innovation and discusses the effects of commercial constraints such as capital and resources on the project. Section 6.5 then discusses the final learning outcomes from the project. The penultimate section (6.6) highlights the impact of the work, and the final section (6.7) overviews the limitations of the work to date.

Please not that the terms ‘identifier naming’ and ‘styling’ will be used interchangeably throughout this discussion; with ‘styling’ being a short-hand for ‘identifier naming’.

6.1. Claim of Innovation

For the purposes of this discussion, innovation will be defined as the *successful exploitation of new ideas, methods or processes* [43]. The innovation claimed for this project, in the context of autocoding methods, is:

- **The integration of PHP scripting with programmer-level XML source code descriptions for the autocoding process, providing user customisation of code formatting, identifier naming, and dynamic autocoding functionality.**

6.2. Autocoding Method

The prototype autocoding platform developed during this project implemented a new, innovative autocoding method to help support the options desired, and resolve the issues present with the XSLT-based method. This autocoding method was based on the integration of two languages to provide the static and dynamic code template functionality required: **XML** (static) and **PHP** (dynamic).

XML was used to describe the static source code contained within the source code templates (which were also implemented using XML). These 'XML code descriptions', in conjunction with the platform's processing methods allowed the source code generated from these descriptions to be customised in terms of its formatting and styling. The use of XML also allowed the platform to automate the selection of static code based on the user's selected compiler and target options. This method will be described further in Section 6.2.1.

PHP scripts were used to fulfil the dynamic requirements of the code templates. These PHP scripts were integrated with the static XML code descriptions and, again using the

processing methods provided by the platform, allowed the code template to compute the output source code required based on the user's inputs. This method will be described further in Section 6.2.2.

6.2.1. Static Autocoding with XML Code Descriptions

The autocoding platform was built around the concept of *Autocoding Applications*. Each autocoding application generated source code for one particular purpose. The user could install additional applications as required, and later select between them through the platform's GUI. The platform would then generate the source code for the user's selected application.

Each autocoding application was contained within an XML file, referred to as the *Application XML* file. This XML file contained 3 key areas of data:

1. Information about the application, e.g. Name, Description, Developer, Version, etc…
   - This information was extracted by the platform and presented to the user to aid selection of their required application.
2. Autocoding options and settings e.g. Supported Languages, Compilers, Targets, Code processor type, etc…
   - This information was extracted by the platform and displayed to the user to enable them to select between the options available for the application selected.
3. Source code file templates
   - These contain the source code templates which are used by the platform and with the various inputs to generate the source code required.

The source code file templates contain the static (XML Code Descriptions) and dynamic (PHP scripts) information for generating the source code for a single output source code

file, with each file being represented as an XML '`File`' element. All possible files that could be generated are stored within the Application XML.

As mentioned, the static code templates are written using XML code descriptions. Each XML code description fully describes a single part of the code, with each one being structured according to the item being described. This method of describing code using XML can best be explained using an example that follows:

Assume that the static code template required a definition of a variable called 'myVariable'. This definition would be output to the source code file as follows:

```
unsigned int myVariable = 0; // A description of my variable
```

Using the XML code description method, this variable definition would be represented by the following:

```
<Variable subtype="_BASIC">
  <StorageClass />
  <DataType>unsigned int</DataType>
  <Identifiers>
    <Identifier id="0">myVariable</Identifier>
  </Identifiers>
  <InitialValue>0</InitialValue>
  <Description>A description of my variable</Description>
</Variable>
```

As can be seen above, XML is used to fully describe the variable definition. The description contains tags for describing the: storage class, data type, identifier, initial value, and description.

An XML source code description is specified for every source code construct that may be contained within a source code file, for example: variables (shown), loops, conditionals, constants, pre-processor directives, structures, arrays, functions, etc. In

total, the prototype autocoding platform and the associated processors support 49 different XML source code descriptions.

An entire source code file can be represented using these XML source code descriptions. The platform's processors progress through each XML code description sequentially from top to bottom (and within the XML hierarchy as required); transforming these descriptions into the formatted and styled source code. The technique for transforming the XML code description to the final source code consists of three basic processing stages: *XML code description selection* (for compiler and target); *code formatting*; and *identifier naming*. These are described below.

6.2.1.1. XML Code Description Selection

One of the requirements of the autocoding platform, based on the listed recommendations, was that the autocoder should support different compilers and targets. For this to be possible, this functionality needed to be further supported by the autocoding method used.

To achieve this, the autocoding method used XML attributes to describe each XML code description's compiler and target compatibility. An example of these attributes is shown below.

```
<Variable ... compiler="Keil|Tasking" target="LPC21XX">
  ...
</Variable>
```

Any XML code description element can contain zero, one, or both of the following attributes: `compiler` and `target`. The platform's processor uses these attributes, along with the user's compiler and target selections to determine whether a static XML

code description should be included in the output or not. If an attribute is missing then it implies that the element is required regardless of the compiler or target selected by the user. In addition, so that an element can be compatible with more than one option, each attribute may contain more than one value. This is achieved by separating the values in the attribute with the '|' symbol.

6.2.1.2. Code Formatting

With the compatible static XML code descriptions selected, the first stage in transforming the XML code descriptions into the output source code was to create the formatted code from the descriptions. This transformation method was based on user editable format strings; with there being one format string for each of the 49 XML code descriptions possible.

Each format strings used a simple mark-up method, where every element of an XML code description had its own mark-up, i.e. there would be a mark-up for the description, data type, storage class, etc. Through the user interface (Figure 19), the user is able to modify this formatting string to suit their requirements. This customisation involves the modification of whitespace, vertical and horizontal space, and the removal of non-required mark-ups, e.g. descriptions. An example of a format string, continuing the variable definition example presented previously, is shown below:

```
// %DESCRIPTION%
%STORAGE_CLASS% %DATA_TYPE% %IDENTIFIER% = %INITIAL%;
```

For every XML code description in the code template the platform's formatting processor replaces the format string mark-up for the item in question with the values contained in the XML code description. Once all values have been replaced the

processor is left with the formatted code string representation for that element. This string is then combined with any previously formatted code until all of the XML code descriptions have been processed. The processor then holds a customised, consistently formatted source code file string based on the user's formatting requirements.

6.2.1.3. Identifier Naming

Although it was stated above that the processor replaces the format string mark-up with the value from the XML code description, this is not entirely true for identifiers within the code. For identifiers, the mark-up is replaced by a second type of mark-up which is used to reference the identifier element contained within the XML code description. An example of this mark-up is as follows:

```
%REF_VAR_LOCAL(0,1,2)%
```

The 3 numbers in the mark-up refer to the File, Item, and Identifier IDs of the XML code description being referenced. After a formatted source code file has been created in memory, the platform's naming processor uses these references to locate, style, and replace the identifier based on the user's settings.

The identifier naming function also uses a formatting string which can be customised by the user. Instead of customising the whitespace, etc, however the user uses a third type of mark-up to select the naming convention used. The user may also add constant characters or strings to the mark-up if required. Examples of these format strings are as follows:

`[ID_CAMEL]`	e.g. MyVariable
`[ID_PASCAL]`	e.g. myVariable
`[ID_CAPS_NO_SEP]`	e.g. MYVARIABLE
`func[ID_CAMEL]`	e.g. funcMyFunction

A formatting string exists for every identifiable item within a source code file. The naming processor iterates through each possible identifier, performing the following operations:

1. Locates the identifier, written in English in the referenced XML code description;
2. Retrieves the format string for the current identifier type and formats the identifier accordingly;
3. Replaces all instances of the identifier reference with the formatted name created in (2).

Once all of the identifier references have been processed, the platform's processor is left with a formatted and styled source code file; the formatting and styling of which is consistent throughout the file (and all other files) and has been customised to the user's requirements.

6.2.2. Dynamic Autocoding with PHP

As described in Section 2.1, the static sections of the code templates are represented using XML code descriptions. The platform's processors use these descriptions to generate the formatted and styled source code. However, using XML code descriptions alone does not allow the source code to be customised based on the user's inputs since all of the templates are static.

The autocoding method uses PHP scripting to provide the dynamic customisation of the generated source code based on the user's inputs. These PHP scripts are integrated with

the application XML and, more specifically, the XML code description method already described.

One of the advantages of XML is that additional data which is not formatted in XML can also be stored within the tags. This is achieved using a special XML tag called CDATA. Specifically, the purpose of this tag is to prevent the XML parser from parsing any data contained between them.

Dynamic sections of the code templates are stored within `Dynamic` elements that can reside anywhere within the main content of the application XML file. Within each `Dynamic` tag is the CDATA tag, which in turn contains the dynamic PHP script. An example of a dynamic code element is as follows:

```
<Dynamic>
  <![CDATA[

  $i = $anotherVar + $i;

  if($i != 0)
    return $xmlCodeElement;
  else
    return NO_ELEMENT_REQUIRED;

  ?>
  ]]>
</Dynamic>
```

The platform's processors iterate through each code template, searching for the dynamic tags and processing them as required. This dynamic processing is all performed before the code formatting and styling is performed (for reasons that will become apparent). When the platform reaches a `Dynamic` element it extracts the PHP script contents. This script is then evaluated using PHP's built in `eval()` function.

Due to the XML code description-based autocoding method used, the integration of the PHP needed to be compatible with this method to work, i.e. any code that is required must be represented using the XML code description format. Every PHP script evaluated, therefore, can return one of 3 results:

- **NO_ELEMENT_REQUIRED**
  - This is a pre-defined constant that is returned when no XML code description is required. The original `Dynamic` element is removed from the XML tree.
- **DOMNode**
  - This represents a single XML code description, for example, a function or a variable description. This DOMNode (i.e. element) is inserted into the XML tree; replacing the original `Dynamic` element.
- **Array of DOMNodes**
  - This is an array of XML code descriptions which are sequentially inserted into the XML tree; replacing the original `Dynamic` element.

Note that the script may also return null (if the script was invalid) and false (if there was an error during the PHP script's calculations).

The platform's processor iterates through the code file template until there are no further `Dynamic` elements present. The platform is then left with a completely static XML code description-based code file template, which can then be used to generate the custom formatted and styled source code using the methods described previously.

6.3. Innovation Justification

This subsection will now justify the claim of innovation presented previously in Section 6.1, and repeated here for convenience:

- **The integration of PHP scripting with programmer-level XML source code descriptions for the autocoding process, providing user customisation of code formatting, identifier naming, and dynamic autocoding functionality.**

The justification will focus on the two main aspects of innovation: Successful Exploitation (Section 6.3.1) and Originality (Section 6.3.2).

6.3.1. Successful Exploitation

The evaluation described in Section 5 and presented fully in Submission 6, implemented a case study that tested out the autocoding method developed on a real application. The case study showed that the autocoding method can be used successfully to implement a powerful and effective autocoding application.

Due to the proper design, implementation, and testing stages performed, the project has not only developed and proved the autocoding method used; it has also resulted in a fully functional autocoding platform. In the platform's current state, it can be used immediately by Rapicore to replace the XSLT-based method currently used by NetGen for their internal autocoding requirements. Once the platform has been integrated, the additional functionality and benefits the new platform provides to the various users of the system are summarised in Table 18.

Users	Identifier naming customisation
	Code formatting customisation
	Language, compiler, and target selection
	User configuration portability
	Process status and feedback
	Individual options for each generation
	Multiple application management and selection (including information on each application)
	Robust and reliable output
	Support for any application complexity level
Developer	Full computational ability
	Large library support
	C-styled programming through PHP
	Variable and constant support
	OOP development
	Cross template variable support
	Pre-processing calculations
	Basic and advanced application support
	Autonomous naming and formatting processing (advanced applications)
Rapicore	Use of the platform with other tools through Host GUI component
	Features that exceed those currently available in other tools
	Plenty of scope for expandability
	Cost effective through the use of open source software
	Cross-platform compatibility

Table 18 - Prototype autocoding platform benefits

At the time of writing, Rapicore is already planning on porting the autocoding applications currently implemented using XSLT to the new XML and PHP-based method for internal code generation applications. The company is also planning to use the autocoding platform for other tools in addition to NetGen, for example, the use of the platform as a standalone autocoder for generating signal-based API stacks that link in with a standardised Hardware Abstraction Layer (HAL) interface.

To summarise, the autocoding method has/will successfully:

- Solved the issues with the XSLT-based method;
- Provided additional functionality and features;
- Generate customised source code to meet the user's development requirements;
- Be commercially exploited by the company in current and future products.

6.3.2. Originality

This subsection will discuss the originality of the autocoding method proposed in this report. Although there is a single claim of innovation that consists of both PHP and XML code descriptions, for the purposes of this discussion the originality of the XML code descriptions and the integration of PHP will be justified individually.

6.3.2.1. XML Code Description

The use of XML to describe source code in itself is not a new concept. However, it is claimed that the use of programmer-level XML source code descriptions for the autocoding process, and its use to generate custom formatted and styled source code has not been done before.

There are a number of methods which use XML to represent source code for the purposes of compiler implementations. These use XML to create an Abstract Syntax Tree (AST) and/or Abstract Syntax Graph (ASG) which are then used by the compilers to generate the required machine code. Such languages include GXL [44], CppML [45], ATerms [46], and Harmonia [47]. However, unlike the use of these in the platform, these representations are intended as data exchange languages or for displaying program structural information. An AST typically represents small grammatical aspects of the source code rather than representing programming-level constructs directly; therefore these methods are not appropriate for autocoding applications.

The most closely related works to the research and developments presented in this report are srcML [48] and JavaML [49]. These use XML source code descriptions primarily for the analysis of source code using the abundance of XML tools and

techniques available. The emphasis with srcML is in combining text with both structural and textural information of the source code. The aim has also been to preserve semantic information from the source code, such as code formatting (the opposite to the purpose used in the platform).

JavaML is similar to srcML and is aimed at describing Java source code in particular. Unlike srcML, JavaML discards formatting information. The XML documents are then used with an XSLT template to display the source code to the user. Again, the focus of the work here seems to be on source code analysis; not on source code generation.

It is also believed that the autocoding method implemented by the platform provides the most customisable level of source code of the other tools researched during the literature review (Section 2). Unlike the other tools, the method is able to customise the code's formatting down to the expression level, and provides full customisation of the identifiers used – features not found in other tools.

6.3.2.2. PHP Scripting

Through the literature review, it was found that no other autocoding tool available uses PHP for the autocoding process; let alone the integration of PHP with XML code descriptions. The languages that were used included: XSLT, TLC (Mathworks), C#, JScript.NET, VB.NET. A number of tools can generate PHP script; but none were found that used the language for the autocoding process itself.

PHP was selected in particular, based on a set of defined requirements that were believed to best suit the autocoding process based on the issues identified with XSLT

and past experience (Section 4.1). What has been proven in this project is that PHP can be used successfully to develop an autocoding platform and for performing the dynamic autocoding processing required. There is no evidence to suggest that this method has been researched or commercially exploited prior to this project.

6.4. Effects of Commercial Constraints

Like all organisations and the projects they undertake, Rapicore and this research project were subject to commercial constraints which had significant impacts on the research and development undertaken. The commercial constraints which had the most significant impacts on this project in particular included: capital, resources, and market. More details on what these constraints were and how they affected the project are discussed in the subsections that follow.

6.4.1. Capital Limitations

Rapicore is a small enterprise and therefore has limited funds for the projects they undertake. This lack of funds is continually restricted by the relatively small market into which a niche product such as NetGen can be sold and therefore generate revenue for the business. The limited funding meant that it was infeasible to consider the purchase of expensive technologies or tools for use during the research and development activities. This, for example, severely impacted the tool research undertaken during the project, as third party tool access was limited, and the research had only existing research papers and tool datasheets to work with.

Fortunately, there is an abundance of open source tools and tools with free licenses

(such as the GNU General Public License). The lack of available funds and the availability of these freely available tools and technologies were the driving factors during the research project and significantly influenced the choices made throughout.

It is widely recognised, however, that the risks to a project are increased when open source tools are used over those that have commercial backing. These risks include: a lack of technical support; abandoning of a project by the voluntary contributors; indeterminate or unreliable time scales for new releases and bug fixes. These factors were considered during the research and development, where the aim was to only utilise tools and technologies that were in widespread use and had plenty of support from the open source community (XML and PHP being prime examples of such technologies).

6.4.2. Resources

Although the word 'resources' covers a number of aspects, the most significant resource limitation during this project was that of human resources. With Rapicore being a small company with limited funds, the increasing of its work force to cater solely for this project was not possible. This lack of human resources considerably affected the project in a number of ways:

- **Peer Reviewing** - The implementation of a peer review process for reviewing the requirements, design, and coded implementation was limited. This reduced the perspective available and resulted in a number of omissions and design issues;
- **Testing** - There were no resources available to independently write the unit, integration, system, and UA tests. To further increase the consequences of this

imposed limitation, there were no independent people available to perform the tests themselves.

6.4.3. Market

Autocoding products, especially those for networked embedded systems sit in a niche product area. Because of this, the field of autocoding methods research is small. Firstly, this fact significantly limited the number of tools against which the developed autocoding platform could be benchmarked. Secondly, the amount of research literature (both quantitative and qualitative) was minimal when compared to areas of more extensive and widespread research. This made the finding of information difficult and was a significant limitation during the literature review in particular.

6.5. Final Learning Outcomes

Although the autocoding platform and new autocoding method was successfully implemented, the project still provided a significant amount of learning which can be taken forward and applied to future projects. This subsection will summarise the learning gained during this research and development project.

6.5.1. Peer Reviewing

The research project demonstrated the importance of the peer review process in software development. This peer reviewing should be used wherever possible, and could conceivably be utilised in all initial stages of the development. For example, requirements should be reviewed to check that mandatory requirements are indeed mandatory, and that requirements do not contradict one another. Peer reviewing should

be used during the design stages to ensure that the design meets the requirements, and any technical limitations (which would require expensive future rework) are not overlooked. Finally, peer reviewing should be used for the implementation stage in the form of code reviews. This ensures that the code fulfils the design and minimises the risk of unforeseen run-time bugs that would be more difficult to identify and rectify in later stages of the project.

6.5.2. Requirements Selection

The project demonstrated how important it is to ensure that the requirements are appropriate and suitable for the project in question. In this instance, the creation of documentation, for example, should not have been defined as a mandatory requirement for a prototype. Doing so and not fulfilling the requirements reflects badly on the project, and could have been easily avoided. Again, a thorough peer review (as discussed above) would have found this; resulting in the requirement being removed or lowered in importance.

6.5.3. Model Selection

At some points during the development, the selection of the V-Model and it's appropriateness for this project came into question. The main point was during the testing stages, where the tests that were developed earlier in the project did not match the final implementation.

Agile and iterative based methods were considered, and would have been equally suitable in hindsight. However, one persuasive factor for using the V-Model was the

requirement to explain the development for this document and elsewhere in the portfolio submissions, i.e. explaining a more linear development such as that created when the V-model is used is easier to understand than that of an iterative development.

6.5.4. Centralised Component Design

In general, the prototype autocoding platform's design and implementation was a success. However, it was noted that there were some design omissions that, if implemented, would have improved the maintainability and expandability of the platform.

Language support should have been centralised. Other components could then query a centralised component to get the details they require. This was not done, and meant that whenever a new language is added to the autocoding capabilities, changes would need to be made in 4 or 5 other components. This increases the risk of omissions and/or mistakes in the modification process and requires the re-testing of existing components.

The learning to be taken forward here is that, if an application has a number of different but related components, the inclusion of the components should be limited to a single component. This significantly improves the expandability of the software, eases maintenance, and reduces the need for the repeated testing of existing, modified components.

6.6. Impact of the Work

Without a doubt, the new prototype autocoding platform will provide significant

benefits to Rapicore; both now and in the future. The platform has significantly improved their autocoding capabilities. Their XSLT-based system, although effective for simple applications, was unable to meet the requirements of more advanced applications; resulting in a loss of business and damage to their reputation (the FlexRay project being a prime example of this impact).

The new autocoding platform has now put Rapicore in a stronger position, allowing them to be more competitive in terms of the autocoding functionality and the options they can provide to their customers.

As highlighted in the introduction, the aim of the project and the platform was to improve their autocoding capability; not to provide them with a new marketable product. However, due to the inclusion of standalone product requirements, and the consideration of these requirements during the design and implementation, Rapicore now have a tool that, with some additional development effort could realistically be retailed as a stand-alone autocoder for a range of different applications. This will help increase revenue, move the company into new markets, and improve the business opportunities available; all of which were not possible with the XSLT-based method.

At present, being a prototype, the tool is suitable for aiding internal developments for customers only. For example, when a customer requires some code for their application, and not the ability to generate the code themselves, Rapicore's developers can use the autocoding platform to generate some or all of the code required. The developer's could easily customise the tool to meet any internal code generation requirements and even use it as a standalone tool, if required.

6.7. Limitations of the Work

It is recognised that there are a number of limitations to the work done for this project. The tool has been developed explicitly as a prototype; not as a production ready tool. The aim was to essentially provide a proof of concept so that the new autocoding method and the platform's design could be tested before any further investment was made.

The work has, without question, proved the effectiveness of the autocoding method and the platform's design. However, there are a number of activities that need to be done before the tool could be considered for official release to Rapicore's customers. These activities are briefly summarised below and discussed in more depth (with approximate timescales) in the Future Work section (Section 8).

Component redesign and implementation

Due to the lack of peer reviewing and the benefits of hindsight, there are a number of components that need to be redesigned and implemented; the most notable being the language support components discussed previously.

Testing

Before the autocoding platform can be considered a production ready tool, the prototype autocoding platform needs to go through further, more extensive testing. This testing needs to include: static code analysis; the development of automated unit and system tests; the performing of the UATs by a user of the system.

Documentation

Full documentation needs to be created for the autocoding platform. This documentation needs to include internal documentation which explains how to develop components for the platform and how to create the XML/PHP code templates. There needs to be a user manual which explains how to select an application, configure the autocoder, and generate the source code. The tool also needs to generate the documentation for the source code created by the platform; taking into consideration the extensive customisation that can be applied to the code.

Template Development Tools

During the platform's evaluation it was shown that the development of the advanced code templates was time-consuming and difficult when compared to that of the basic templates. A tool therefore needs to be developed to reduce the time and difficulty required to develop the advanced XML/PHP autocoding templates.

7. Conclusion

In conclusion, this project has resulted in the research and development of a new prototype autocoding platform which utilises a more suitable autocoding method. All aspects of the platform were researched, designed and developed; learning from the limitations of the XSLT-based method currently used by NetGen. Other autocoding methods and tools were researched to ensure that the platform met Rapicore's customer's expectations in terms of the autocoding functions, features and options the tool provides.

The autocoding platform has successfully proved that PHP, together with the GTK+ toolkit can be used to develop a powerful autocoding platform that meets the requirements and expectations of such a tool. A new, original autocoding method using XML with integrated PHP has been conceptualised, designed and developed. This unique method has been integrated into the platform; allowing the user to configure and customise the autocoding process and the source code generated.

The language testing and platform evaluation demonstrated that both the platform and the XML/PHP autocoding method can effectively generate source code for applications with complex computational requirements, and that this method successfully solves the issues identified with the XSLT-based method.

As the platform has been designed with versatility, expandability, and maintainability in mind from the start, the autocoding platform will support Rapicore's autocoding requirements into the future. In addition, the autocoding platform and autocoding

method provides a range of functions and features that were not present or possible with the XSLT-based method. These are summarised in Table 19.

Autocoding Platform	• Host customisation, i.e. for using the platform with tools other than NetGen;
	• Installation and selection of additional autocoding applications;
	• Code customisation in terms of:
	  o Processor;
	  o Compiler;
	  o Language;
	  o Styling and formatting.
	• Generation of code with different options;
	• Gathering of application specific settings from the user;
	• Robust error checking and feedback of information to the user.

XML/PHPAutocoding Method	• Complete mathematical processing functionality; • Full set of computational constructs, e.g. loops, conditionals, data types; • Modular and structured implementation with XML; • Reusable template components; • Simple and advanced template implementations to suit the developer and/or project requirements; • Easily expanded.

Table 19 - Platform and autocoding method benefits/features summary

As has been made clear throughout this report, Rapicore now have a prototype autocoding platform which is suitable for their internal code generation requirements, only. It is ready to be integrated with internal versions of NetGen to generate code for consultancy-based development projects. The tool can also be used as a standalone autocoder and customised to generate code for other internal application requirements.

The autocoding platform now needs to be prepared for its release into a production environment. Once this has been achieved the autocoding platform will be used immediately with NetGen; replacing the current XSLT-based method. Existing XSLT-

based autocoding applications and their associated templates will be ported over to the new autocoding platform. Rapicore also have the option of releasing the platform as a standalone autocoding application; allowing them to move into a more generic software market, increasing their product portfolio and creating an additional source of revenue.

The research project and the various research and development stages undertaken have provided some significant learning opportunities. The project has demonstrated the need for a thorough peer review process throughout the requirements capture, design, and implementation stages of a software development. The process has also proved how the commercial constraints of funding, resources, and market can impact a project.

If the project were to be undertaken again, the key recommendation based on the experience and learning gained would be to seek a commercial partner that could provide additional funding and resources. This would facilitate the purchase of additional tools and technologies; widening the scope of the research and development. The partner could also provide additional human resources so that the various peer reviews could be performed; minimising errors, omissions, and the need for future rework.

8. Future Work

This research project has produced a fully functional autocoding platform prototype. Although the development of a prototype to demonstrate the autocoding platform and associated methods was the original aim of the project, the fact that only a prototype exists at present is recognised as a limitation of the work and the platform's future commercial value.

The aim of the future work is to ready the prototype autocoding platform for its release into a production environment. For this to happen, a number of design, development, documentation, and preparation activities need to be performed. This final section will explain the activities and the approximate timescales needed to perform them before the autocoding platform can be released. These activities are split into short and long term activities. The short term activities relate to the work that needs to be done for release, and the long term activities are optional; relating solely to the future direction of the tool.

8.1. Short Term

Component Redesign and Implementation

One of the learning points during the platform's development was to centralise the language support capabilities within a single component. The other components could then query this centralised component to determine which languages are currently supported by the platform. Therefore, a single language component needs to be designed, implemented, tested, and integrated into the platform. In addition, the other components that require knowledge of which languages are supported need to be modified so that they can query this component.

The C language components (`CFormatting` and `CNaming` classes) need some minor redesign to more robustly fix some issues found during testing. The main change is to provide a means of overriding the custom naming of the generated source code's 'main' function (and any other identifiers required). This is the entry point for the program and must always be called 'main' and be in lower case. In addition, all of the language formatting component's internals need to be checked to ensure that they generate code that can be compiled. No instances where found where this was not the case, but additional testing to more thoroughly confirm this functionality is recommended.

Finally, every component within the system needs to have a testing mode added to it. This aims to improve the platforms "design for testability", making it easier to develop effective and reliable unit tests.

Static Code Analysis

At present, the unit, integration, and UA tests have been performed but no static analysis has been carried out on the platform's code. This analysis will provide software metrics, helping identify any unseen bugs within the code, as well as providing information with which the code can be improved, e.g. the identification of unused variables, or functions that have not been executed. PHP-sat is one such tool that would be suitable for this task [50].

Continuous Integration Environment

Once the platform's components have been redesigned, implemented, and tested using both static analysis and dynamic testing, the next activity is to setup a continuous

integration (CI) environment. This will consist of a number of sub-activities which are the setting up of a:

- ***Version control system*** (e.g. Mercurial [51]) for submitting and tracking code changes;
- ***Bug and Feature tracking application*** (e.g. Redmine [52]) for recording the identification and resolution of bugs and feature requests;
- ***Automated build system***. As PHP is interpreted, the term 'build' (which relates more to compiled languages) would refer to the configuration and collection of the various components that make up the platform. This includes the configuration of different versions of the platform, e.g. trial versions, cut-down versions, fully featured versions, etc;
- ***Automated testing environment*** so that the platform and its components can be continuously tested as new code is added and existing code is changed. This automated test environment would consist of automated unit tests (using the likes of PHPUnit [53]) and automated system tests.

XML/PHP Template Development Tool

As discussed, the development of the advanced applications for the platform is a relatively time consuming and difficult task. In order to improve productivity and reduce the risk of bugs, a tool needs to be developed to aid the process. The envisaged features and capabilities of this tool would be:

- To automatically convert existing source code into static XML Code Descriptions;
- Allow developers to insert XML code descriptions from a library (as opposed to creating them all manually or copying and pasting them from another

document);

- Monitor and keep track of variable, constant and function references to minimise errors, e.g. referencing the incorrect variable in a function;
- Provide a means of validating the application XML to ensure that it conforms to the platform's requirements.

Of course, some documentation that describes how to use the tool to develop the code templates will also be required.

Documentation

At this point, the platform will be almost ready for initial release. However, at present there is no documentation for internal or external audiences (other than that contained within the portfolio submissions) that instructs users how to use the platform. Three documents (or collections of documents) need to be developed:

- ***Platform Development and Maintenance Manuals*** – these documents would be used by developers at Rapicore to add new features and/or components to the platform;
- ***Application Development Manual and References*** – these documents would again be used by Rapicore's developers to create the autocoding templates for the customer's custom or off-the-shelf applications;
- ***Platform User Manuals*** – these documents would be made available to the users of NetGen and would instruct them how to use the autocoding platform to generate the code they require. It would detail how to install and select applications, how to configure the platform, and then how to generate their code.

Port XSLT-based templates to the platform

All of current XSLT-based templates must then be ported to the platform's autocoding template format, ideally into the advanced application format to provide the most benefits to the customers. With the development of the application development tool mentioned above, and the relative simplicity of these templates this should not take too long.

Integration of the platform into NetGen

During the development the platform has been run as a standalone application from the command line, however, the integration of the platform with NetGen was considered during the design. The XSLT-based method must first be removed from NetGen. Then NetGen must be modified to call the platform (the method of which is very similar to the command line execution currently used).

Beta Testing Period

It is recommended that the first release of the autocoding platform to the public be done as a beta release to a small number of select customers. This will test out the platform in a real, but limited number of production environments, making the collection of feedback and the roll out of any fixes and new features more manageable.

Full Release

Once the platform has been tested, any unforeseen bugs have been fixed, and any additional functions and features desired by the customers have been implemented and tested, the platform will then be ready for full release.

8.2. Long Term

Develop new autocoding applications

In the long term, Rapicore should aim to develop a range of new applications for the platform which are appropriate for the current business environment and market. For example, Rapicore could look into developing autocoding applications for additional network protocols such as Ethernet and MOST.

Develop additional language support

The platform has been designed so that it can support more than one output language, but at present it only supports C. In the long term, Rapicore should look into developing the language components for other languages. Recommended languages would be C++ and Ada based on their popularity during the literature and tool review stages of this project.

8.3. Activity Times

Table 20 below shows the best, worst and average predicated times for the Short Term activities described in Section 8.1. The best case time assumes that the activity is carried out without any issues. The worst case time assumes that there are significant difficulties during the activity, and the average time is the average of the best and worst case times. Note that these are guideline times only. Long Term activities have not been included here due to the variable nature of those in question.

As can be seen from the table, using the average time estimate, the autocoding platform could be ready for release in 327.5 man days. Some of the tasks can feasibly be run

concurrently, meaning that the autocoding platform could be ready for release within a year of starting the first activity.

Activity	Implementation Times (Man Days)		
	Best Case	Worst Case	Average
Component Re-designs	15	25	20
Static Code Analysis	5	7	6
Continuous Integration and Testing Environment	10	30	20
Application Development Tool	90	200	145
Documentation	30	35	32.5
XSLT Porting	10	15	12.5
NetGen Integration	1	2	1.5
Beta Testing	90	90	90
Estimated Time to Release	**251**	**404**	**327.5**

Table 20 - Timescales for the future work activities

References

[1] Strunk, E.A., Knight, J.C. & Aiello, M.A., *'Distributed Reconfigurable Avionics Architectures'*, Digital Avionics Systems Conference (DASC), Volume 2, p101-110, 2004.

[2] Bauer, F.L., Bolliet, L. & Helms, H.J., '*Software Engineering'*, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October, 1968. Editors: Naur, P. & Randell, B.

[3] Moore, G.E., *'Cramming more components onto integrated circuits'*, Electronics, Volume 38, Number 8, 19th April 1965.

[4] Goldman, T., *'Educating the Next Set of Developers'*, presentation for Philippine Open Source Summit, CICC, June 23-24 2008. Available online at: http://oss.ph/files/downloads/IBM_Educating_the_Next_Set_of_Developers.pdf

[5] IEEE, *'IEEE Standard Glossary of Software Engineering Terminology'*, IEEE STD 610.12-1990, 1990.

[6] Fisher, A.S, '*CASE: Using Software Development Tools',* 2e, Wiley, 1991.

[7] Standish Group, *'The CHAOS Report 2004'*, Standish Group, 2004.

[8] OMG, *'Unified Modeling Language'*, UML Resource Page, Object Management Group, Available online at: http://www.uml.org [Last accessed: 23rd September 2009]

[9] Rapicore, *'Rapicore Ltd Home Page'*, Official website, Available online at: http://www.rapicore.net. [Last accessed: 23rd September 2009]

[10] W3C, *'World Wide Web Consortium – Web Standards'*, Official W3C website, Available online at: http://www.w3.org. [Last accessed: 23/09/09]

[11] W3C, *'XSL Transformations (XSLT) Version 2.0'*, W3C Recommendation, World Wide Web Consortium, 23rd January 2007, Available online at: http://www.w3.org/TR/xslt20/#dt-processor [Last accessed: 23rd September 2009]

[12] Finney, J.G., *'Software Engineering and Code Generation for Embedded Systems'*, Engineering Doctorate portfolio 1st Submission, August 2007.

[13] The MathWorks, *'Control Design: Description Topics: Model-Based Design'*, Available online at: http://www.mathworks.co.uk/applications/controldesign/description/index.html [Last accessed: 29th September 2009].

[14] The MathWorks, *'The MathWorks: Simulink'*, Official tool page, Available online at: http://www.mathworks.co.uk/products/simulink/ [Last accessed: 28th September 2009].

[15] Esterel Technologies, *'Esterel Technologies: SCADE Suite'*, Official tool webpage, Available online at: http://www.esterel-technologies.com/products/scade-suite/ [Last accessed: 29th September 2009]. (2009, a)

[16] ETAS, *'ETAS: ASCET-MD'*, Official tool webpage, Available online at: http://www.etas.com/en/products/ascet_md_modeling_design.php [Last accessed: 29th September 2009].

[17] Finney, J.G., *'Model-Based Design and Autocoding Tools'*, Engineering Doctorate portfolio 2nd Submission, April 2008.

[18] The MathWorks, *'The MathWorks: Real-Time Workshop'*, Official tool page, Available online at: http://www.mathworks.co.uk/products/rtw/ [Last accessed: 28th September 2009].

[19] The MathWorks, *'The MathWorks: Stateflow Coder'*, Official tool page, Available online at: http://www.mathworks.co.uk/products/sfcoder/ [Last accessed: 28th September 2009].

[20] dSPACE, *'dSPACE: TargetLink'*, Official product webpage, Available online at: http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm [Last accessed: 29th September 2009].

[21] Esterel Technologies, *'DO-178B Code Generation'*, Official tool webpage. Available online at: http://www.esterel-technologies.com/products/scade-suite/do-178b-code-generation/ [Last accessed: 29th September 2009]

[22] Esterel Technologies, *'IEC-61508 Code Generation'*, Official tool webpage. Available online at: http://www.esterel-technologies.com/products/scade-suite/iec-61508-code-generation/ [Last accessed: 29th September 2009]

[23] Esterel Technologies, *'EN 50128 Code Generation'*, Official tool webpage. Available online at: http://www.esterel-technologies.com/products/scade-suite/en-50128-code-generation/ [Last accessed: 29th September 2009]

[24] ETAS, *'ETAS: ASCET-SE'*, Official tool webpage, Available online at: http://www.etas.com/en/products/ascet_se_software_engineering.php [Last accessed: 29th September 2009].

[25] Mentor Graphics, *'Mentor Graphics'*, Official website, Available online at: http://www.mentor.com/ [Last accessed: 29th September 2009].

[26] Vector, *'Vector Group'*, Official website, Available online at: http://www.vector.com [Last accessed: 29th September 2009].

[27] Elektrobit, *'Elektrobit'*, Official website, Available online at: http://www.elektrobit.com/. [Last accessed: 29th September 2009]

[28] TTTech, *'TTTech'*, Official website, Available online at: http://www.tttech.com/ [Last accessed: 29th September 2009].

[29] Finney, J.G., *'An Evaluation of NetGen as a Generic Network Development Tool'*, Engineering Doctorate portfolio 3rd Submission, September 2008.

[30] AUTOSAR, *'AUTOSAR: Automotive Open System Architecture'*, Official website, Available online at: http://www.autosar.org. [Last accessed 29th September 2009].

[31] CodeSmith Tools, *'CodeSmith Tools'*, Official website, Available online at: http://www.codesmithtools.com/. [Last accessed: 29th September 2009]

[32] Budinsky, F., Finnie, M., Yu, P. & Vlissides, J., *'Automatic Code Generation from Design Patterns'*, IBM Systems Journal, Vol. 35, No. 2, pp.151-171, 1996.

[33] Microsoft, *'Microsoft Visual Studio'*, Official tool page, Available online at: http://www.microsoft.com/visualstudio/en-gb/default.mspx [Last accessed: 29th September 2009].

[34] NXP, *'Products: Microcontrollers: LPC2000'*, Device family webpage, Available online at: http://www.standardics.nxp.com/products/lpc2000/ [Last accessed: 29th September 2009].

[35] Murdoch University, *'HOPL: an interactive Roster of Programming Languages'*, Available online at: http://hopl.murdoch.edu.au/ [Last accessed: 20th November 2008].

[36] Finney, J.G., *'Autocoding Language Research and Selection'*, Engineering Doctorate portfolio 4th Submission, February 2009.

[37] IBM, *'IBM Rational DOORS'*, Official tool webpage, Available online at: http://www-01.ibm.com/software/awdtools/doors/ [Last accessed: 29th September 2009].

[38] Finney, J.G., *'Autocoding Platform Development'*, Engineering Doctorate portfolio 5th Submission, June 2009.

[39] GTK+, *'The GTK+ Project'*, Official website, Available online at: http://www.gtk.org [Last accessed: 20th May 2009].

[40] IBM, *'IBM Rational Tau'*, Official tool webpage, Available online at: http://www.ibm.com/developerworks/rational/products/tau/ [Last accessed: 29th September 2009].

[41] Zend Technologies, *'Zend Framework: Programmer's Reference Guide'*, Online user manual, Available at: http://framework.zend.com/manual/en/index.html [Last accessed: 20th May 2009].

[42] Finney, J.G., *'Prototype Autocoding Platform Evaluation*, Engineering Doctorate portfolio 6th Submission, September 2009.

[43] Luecke, R., Katz, R., *'Managing Creativity and Innovation'*, Boston, MA, Harvard Business School Press, 2003.

[44] Holt, R.C., Winter, A., and Schürr, A., *'GXL: Toward a Standard Exchange Format'*, in Proceedings of 7th Working Conference on Reverse Engineering (WCRE'00), Brisbane, Queensland, Australia, November 23-25 2000, pp 162-171.

[45] Mammas, E. & Kontogiannis, C., *'Towards Portable Source Code Representations using XML'*, in Proceedings of 7th Working Conference on Reverse Engineering (WCRE'00), Brisbane, Queensland, Australia, November 23-25 2000, pp 172-182.

[46] Van den Brand, M., Sellink, A., & Verhoef, C., *'Current Parsing Techniques in Software Renovation Considered Harmful'*, in Proceedings of 6th International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26 1998, pp. 108-117.

[47] Boshernistan, M. & Graham, S.L., *'Designing an XML-Based Exchange Format for Harmonia'*, in Proceedings of 7th Working Conference on Reverse Engineering (WCRE'00), Brisbane, Australia, November 23-25 2000, pp. 287-289.

[48] Maletic, J.I., Collard, M.L. & Marcus, A., *'Source Code Files as Structured Documents'*, in Proceedings of 10th International Workshop on Program Comprehension (IWPC'02), Paris, France, 27-29 June 2002, pp. 289-292.

[49] Badros, G.J., *'JaveML: A Markup Language for Java Source Code'*, in Proceedings of 9th International World Wide Web Conference (WWW9), Amsterdam, The Netherlands, May 13-15 2000.

[50] PHP-sat, *'PHP-sat: PHP Static Analysis Tool'*, Official tool website, available at: http://www.program-transformation.org/PHP/PhpSat. [Last accessed: 17th April 2011].

[51] Mercurial, *'Mercurial* SCM', Official tool website, Available at: http://mercurial.selenic.com/. [Last accessed: 17th April 2011].

[52] Redmine, *'Redmine – Overview – Redmine'*, Official tool website, available at: http://www.redmine.org/. [Last accessed: 17th April 2011].

[53] PHPUnit, *'PhpUnit – unit testing framework for PHP'*, Official tool website, available at: http://phpunit.sourceforge.net/. [Last accessed: 17th April 2011].

Appendices

Appendix A – MBD and autocoding tool comparison table

	ASCET-MD	SCADE	Simulink	TargetLink
Tool Type	MBD	MDB	MDB	Autocoder
Internal Integrations	ASCET-SE (Autocoding) ASCET_MDV (Model Viewer) ASCET-RP (Rapid Prototyping) ASCET-DIFF (Model Difference Explorer) RTA-OSEK INTECRIO	DO-178B Code Generator IEC 61508 Code Generator EN 50128 Code Generator	Real-Time Workshop RTW Embedded Coder Stateflow Stateflow Coder Embedded Targets Polyspace Model Link SL	MTest
Third Party Integrations	MATLAB/Simulink Models ARTISAN Studio (UML)	Simulink Rhapsody (UML & SysML) DOORS (Requirements) Requirements Management Configuration management	MTest Embedded Tester DOORS Prover Plug-in	Stateflow Simulink (Mandatory) Polyspace Model Link TL
Model Optimisation	Yes	Yes	Yes	Yes
Code Optimisation	Yes	Yes	Yes	Yes
Embedded Support	Yes	No	Yes (with Embedded Coder)	Yes
Specific Processor Support	Yes	No	Yes (with Embedded Target)	Yes
Technologies	Unknown	TCL*	TLC and XSLT	TLC and XSLT
Code Customisation	Unknown	Yes	Yes	Yes
Simulation Modes	SIL	MIL	MIL, SIL, PIL (with RTW)	MIL, SIL, PIL
Standards/Guidelines	MISRA-C IEC 61508 SIL3	D0-178B IEC 61508 EN 50128 MISRA-C	None	None
Code Languages	ANSI C	Ada & Ada Spark C and Qualifiable C	ANSI C (Standard), Any other language using S-Functions	ANSI C (Standard), Any other language using S-Functions
Testing and Verification	Yes	Yes	Yes	Yes
Tool Cost	£27,145 (MD + SE)	£61,280	£17,450	£10,790 (+ £6350)
RTOS Support	Yes (OSEK)	Yes (MicroC/OS-II)	Yes (OSEK)	Yes (OSEK)
Inclusion of Legacy Code	Yes	Yes	Yes	Yes (in C, Fortran, Ada, C++)
State Machine Support	Yes	Yes	Yes (Stateflow)	Yes (Stateflow)
Common Industries	Automotive	Aerospace Automotive	Automotive Industrial Control	Automotive Industrial Control

Appendix B – Mandatory requirement recordings

Languages	Notes	Requirement Reference																	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Ant	Ant was designed for the software build process. It is XML based and dedicated for this task, meaning that it is not a general purpose language.	✓	✗																
APL	APL is an array-programming language, and lacks any more general programming syntax and capabilities.	✓	✗																
AppleScript	AppleScript, as the name suggests is dedicated to Apple based operating systems, and is therefore not available for Windows OS.	✗																	
AutoIt	AutoIt is an automation language for windows, used for automating tasks such as network management, and is therefore not a general-purpose language.	✓	✗																
awk	awk provide no built in functionality for reading XML files. It can read files generally, but a full parser would need to be created in order to support XML which would be very time consuming.	✓	✓	✓	✓	✗													
BASIC	BASIC has been superseded by Visual Basic, and in many respects thinBasic would be a more appropriate alternative. In addition BASIC is very low level, with no abstract functionality and no XML processing capabilities.	✓	✓	✓	✓	✗													
BeanShell	BeanShell is a more scripted version of Java, but requires Java to operate, meaning that Java itself would be a more appropriate option.	✓	✓	✗															
Ch (C/C++)	Ch is a C and C++ interpreter. Due to the low level of C/C++ it provides little abstraction, including the lack of string based processing (uses character arrays), making it less appropriate for strings than other known languages.	✓	✓	✓	✗														
ColdFusion	ColdFusion is used for creating dynamic web pages, and focuses on components such as forms. It is therefore not a general purpose languages.	✓	✗																
Databus (PL/B)	PL/B (aka Databus) is a business oriented programming language; not a general purpose language. It also has very little use and support compared to many other languages.	✓	✗																
ECMAScript	JavaScript and Jscripts are dialects of ECMAScript. The language is used for embedded in web pages to provide additional functionality over HTML, but outside of a web-browser is requires an embedded processor in NetGen.	✓	✓	✗															
Falcon	Falcon has a very small user base and community, and lacks development support and therefore stability and peer testing.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
Frink	Frink is built on the Java Virtual Machine (JVM), but only adds the tracking and calculation of units for the purposes of science and engineering. It would therefore be more appropriate to use Java.	✓	✓	✓	✓	✗													
F-Script	F-Script is in essence Smalltalk with the addition of array-processing. It is only available for Mac OS and not Windows.	✗																	
Game Maker Language (GML)	GML is a scripting language for an application called Game Maker, and is heavily integrated with the environment, and therefore not a general-purpose programming language.	✓	✗																
J	J is an array programming language, and not a general purpose language.	✓	✗																
JASS	JASS is used solely for game development, and is therefore not a general-purpose language.	✓	✗																
Lua	Lua is distributed in source code form and requires compiling. Many other languages that do not require compilation of the interpreter itself.	✓	✓	✗															
M	M is a database oriented language, not general purpose.	✓	✗																
MAXScript	MAXScript is used for 3Ds Max Studio scripting.	✓	✗																
MEL	MEL is for Autodesk's Maya 3D software tool (not general purpose).	✓	✗																

Mondrian	Mondrian is an internet scripting language with little support, and is therefore not suitable for general purpose applications.	✓	✗																			
Perl		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PHP		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Pikt	Pikt is for Unix and Linux operating systems only.	✗																				
PostScript	PostScript is used for document descriptions such as those used for the likes of Adobe Acrobat, and is therefore not a general purpose language.	✓	✗																			
Python		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Revolution	Revolution is similar to many other language, but is proprietary. It is therefore not as cost effective as other languages available.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗				
Ruby		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Tcl		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
thinBasic	thinBasic does not provide object-oriented programming.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗		
VBScript	VBScript is a form of ECMAScript. Requires ActiveX control to run, as did ECMAScript.	✓	✓	✗																		
Windows PowerShell	PowerShell is a relatively new scripting shell for Windows. It lacks general purpose functionality and requires the full installation of the PowerShell application.	✓	✗																			
XSLT	This is the current language used by NetGen, and is present here as a control value.	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓

Appendix C – Autocoding test findings summary

	Perl	PHP	Python	Ruby	Tcl
Successful	**Yes**	**Yes**	**Yes**	**No**	**Yes**
Benefits	• Module installation client • Perl debugger • Useful block text syntax	• Full output formatting control • Close to C • Functions pre-installed • Scripts included like C • Automatic type conversion • Biggest community support • Intelligent integer testing	• Everything is an object	• Useful and logical additional syntax • Everything is and object • All required functions installed as standard • Debugger available	• 'Teacup' module client • Easy XML implementation
Issues	• Included script paths always relative to Perl executable • XML modules not pre-installed • List and array indexing inconsistent • XPath searches return different types • Poor documentation • Excessive syntactic sugar • No function for integer checking	• Global variable scoping too global • No debugger support	• Arrays not a core data type • Globals have to be implemented as a separate module • No module installation client • Uses indentation for program structure • No switch statement • Cannot import module containing .h or .c • No unary increment of decrement operator • Cannot access globals from included script	• XML DOM not used • Error prone loop syntax • Some global variables are visible and others are not • Output formatting failed	• Limited locations of included scripts • No constants • Conversion of everything to string • Trouble using XML package • Whitespace is important

Appendix D – Optional requirement scores

Requirement	Perl	PHP	Python	Ruby	Tcl	XSLT
C-like						
Statement oriented	1	1	1	1	0	0
Semicolon terminator for statements	1	1	0	0	0	0
Function oriented	1	1	1	1	1	0
Named parameter access	0	1	1	1	1	0
Curley brackets to group code blocks	1	1	0	0	1	0
while loop	1	1	0	1	0	0
do…while loop	1	1	0	0	1	0
for loop	1	1	1	1	1	0
switch	1	1	0	1	1	1
array indexing from 0	0	1	1	1	1	0
Score	8	10	5	7	7	1
Weight	**1.0**					
Weighted Score	8.0	10.0	5.0	7.0	7.0	1.0
Community						
Tutorials (search for "XX tutorial")	51100	256000	74500	19600	3770	11000
Score	2	10	3	1	0	0
Books	909	1495	196	174	2065	92
Score	4	7	1	1	10	0
Forums (search for "XX forum")	36000	6590000	50800	366000	716	10300
Score	0	10	0	1	0	0
Subtotal Score (av)	2	9	1	1	3	0
Weight	**0.9**					
Weighted Score	1.8	8.1	0.9	0.9	2.7	0.0
Distributable size						
Installation size	130	6.31	53.7	140	35.2	0
Max	140					
Score	1	10	6	0	7	10
Weight	**0.8**					
Weighted Score	0.8	8.0	4.8	0.0	5.6	8.0
Readability						
Multiline comments	0	1	0	0	0	1
Free strucuting						
Inside code blocks	1	1	0	1	1	1
Function definitions	1	1	0	1	0	0
Arrays and structures	1	1	0	1	0	0
Does not use "Syntatic Sugar"	0	1	0	1	1	0
Score	3	5	0	4	2	2
Subtotal Score (double)	6	10	0	8	4	4
Weight	**0.7**					
Weighted Score	4.2	7.0	0.0	5.6	2.8	2.8
Dynamic Typing (Score)	10	10	10	10	10	10
Weight	**0.6**					
Wiehgted Score	6.0	6.0	6.0	6.0	6.0	6.0
Total (Unweighted)	**27**	**49**	**22**	**26**	**31**	**25**
Total (Weighted)	**20.8**	**39.1**	**16.7**	**19.5**	**24.1**	**17.8**